intel

ARTICLE REPRINT

AR-101

NOVEMBER 1979

Exploring the 8086: Part One

Associate Editors
EDN Magazine

Exploring the Bible: Part One

Exploring the 8086: Part One

Table of Contents

Reprinted with permission from EDN Magazine, January 1979, Vol. 24, No. 2; February 1979, Vol. 24, No. 3; March 1979, Vol. 24, No. 6; and May 1979, Vol. 24, No. 10; by CAHNERS PUBLISHING COMPANY.

Exploring 16-bit μPs

As you get to know the 8086, use your 8-bit expertise

It isn't enough to have the hardware and manuals in your hand; you also need an objective, a perspective and some tools.

Jack Hemenway and **Edward Teja,**
Associate Editors

You've probably read about the latest 16-bit μPs and are anxious to know how they work, how they can simplify your design problems and what pitfalls their use will present you with. And as both engineers and editors, so were we. So we set out to explore the world of 16-bit micros, utilizing our experience with both larger and smaller machines and passing along what we learned. Our first goal, to be documented in the first few articles in this series, is an inquiry into the operation and use of Intel's 8086.

With an 8086-based SDK-86 single-board system-design kit actually in our hands, we were excited. It's one thing to know that such a creature as a 16-bit μC can exist and quite another to have one, albeit in its unbuilt kit form, in your possession. Needless to say, the kit went together during the first 24 hrs we had it.

Putting the kit to work

As soon as the kit was built, we faced the same difficulty that every EE faces at this stage in μC-system design: How do you make the thing go? By itself, without a defined objective and a means of achieving that objective, the kit is just another electronic paperweight.

Our objective was simple: We aimed to meet the challenge of using our skills in dealing with μCs to make the 8086 do more than just manipulate bits internally. Equally important was the idea of using this opportunity to examine what designs the 8086 suits particularly well and for what uses it would not be the best choice.

The element of perspective—also a key factor in the success of a design project like this one—provides insight into the methods and hardware we used to achieve our objectives. The worktables in our lab are filled with μCs. Why bother with a new one that's bereft of applications software? The answer is simple enough: The 8086 has the potential of being far more powerful and versatile than any of the 8-bit micros currently available.

A third success factor in a 16-bit design project is the availability of the proper development tools. When selecting these tools, you can take advantage of the fact that you (and most engineers in your position) have been here before with 8-bit designs. Developing a 16-bit machine's potential is not exactly the same as exploiting an 8-bit device, but you shouldn't start *from scratch* just because of a change in architecture any more than you would throw away any of your old tools when you start a new project.

Something old, something new

Rather than designing a totally new 16-bit device, Intel has chosen to make the 8086 a logical extension of the 8080. Thus, 8080 users can gradually adjust to the new device. In

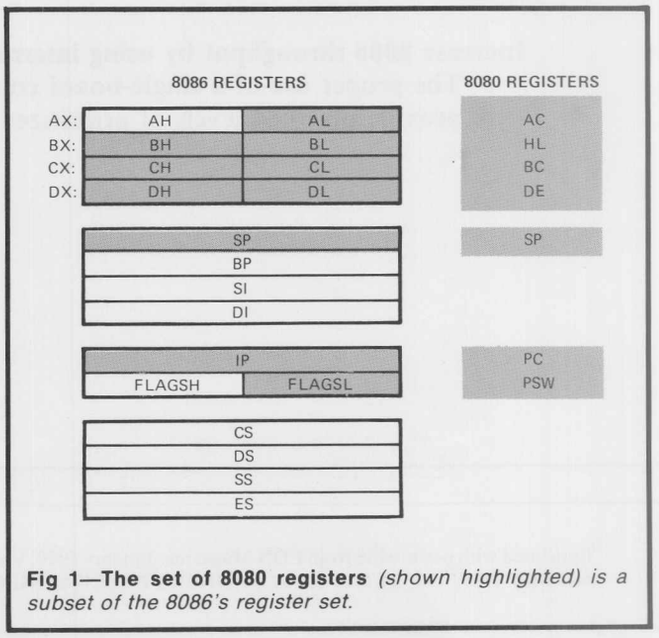


Fig 1—The set of 8080 registers *(shown highlighted) is a subset of the 8086's register set.*

Make use of your skills with 8-bit machines when moving to the 8086

particular, the presence of new instructions and features doesn't imply that you have to put them to work immediately.

It isn't the proper function of this article to define and discuss all the aspects of the 8086. We must necessarily examine particular features related to the tasks at hand, but suffice it to say that the chip is a combination of 8080 features and totally new ones. Thus, spending some time acquainting yourself with the things that have not changed will prove fruitful and could reduce your development time.

Fig 1 illustrates the increased number of registers available to you in the 8086; the highlighted registers are the only ones present in the 8080. And **Table 1** compares the instruction sets of the 8086 and its predecessors. As shown, the 8080 instruction set is an improper subset of the 8086's; the MCS-86 User's Manual provides these and other comparisons that let you put your 8080 knowhow to work on the 8086.

```
; INITIALIZE P1A PORT FOR OUTPUT
; SDK-86 BOARD USING 8255 PARALLEL INTERFACE
; I/O IN TOP SEGMENT, RAM IN BOTTOM SEGMENT
;
        B900F0          MOV CX,0F000H
        8EC1            MOV ES,CX
;
; SET EXTRA-SEGMENT POINTER TO TOP SEGMENT
;
        BAFFFF          MOV DX,0FFFFH
;
; SET DX REGISTER TO CONTROL PORT
; ADDRESS IN SEGMENT
;
        B080            MOV AL,080H
;
; SET AL REGISTER TO OUTPUT CONTROL SETTING
;
      26EE              OUT [ES]
;
; OUTPUT AL TO DX IN SEGMENT ES
;
```

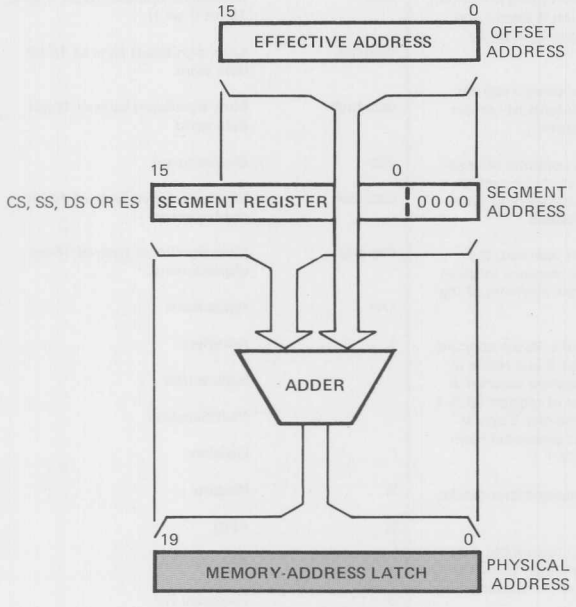
Fig 2—The SDK-86 parallel port, an 8255, requires an 80H in its control register *to set it to accept data for output. This routine passes the byte you wish to output in the low-order byte of the accumulator.*

(a)

```
; OUTPUT A BYTE TO P1A
; PORT INITIALIZED BY THE ROUTINE IN FIG 2
; OUTPUT BYTE IN AL REGISTER
;
        B900F0          MOV CX,0F000H
        8EC1            MOV ES,CX
;
; SET EXTRA-SEGMENT POINTER
;
        BAF9FF          MOV DX,0FFF9H
;
; POINT DX REGISTER TO DATA PORT
; ADDRESS IN SEGMENT
;
      26EE              OUT [ES]
;
; OUTPUT CONTENTS OF AL TO DX
; IN SEGMENT ES
;
```

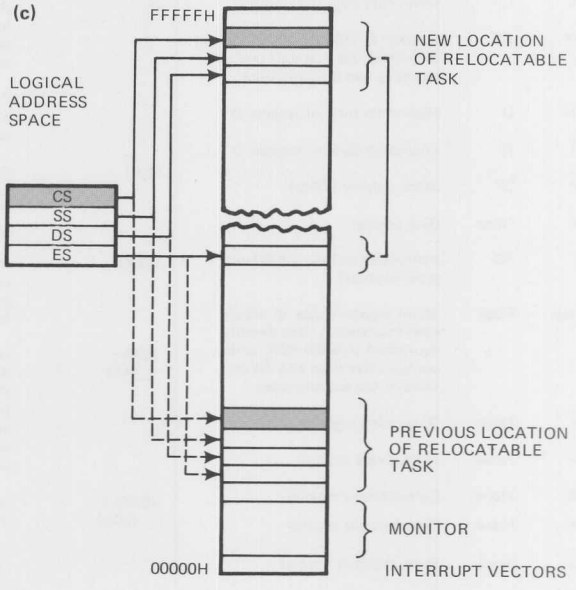
Fig 3—(a) Outputting a byte *proceeds in much the same way that initializing the port does, except that the destination address is now the location of the output port or data register.*

(b)



(b) The actual destination address latched is the sum of the effective (intrasegment) address and the segment displacement stored in the specified segment register.

(c)



(c) Each of the four segment registers can point to a different segment of memory.

One change that's not necessarily for the better is the modification of the conditional jump. In the 8080, such conditional jumps can be made to any 16-bit absolute address. The 8086, on the other hand, provides for relative addressing on a conditional branch (which suits relocatable code) but restricts the offset to +128 and −126 bytes (which can be a problem). This provision means that to branch to locations outside the range of the conditional jump you must resort to unconditional branches—an approach that could force you to recode a conditional branch in a program to add an unconditional one if you have added code to the program between the conditional-jump statement and the jump's target (identified by its label).

The most readily available tool at your disposal, especially when you are just beginning, is the monitor. Two such programs are provided with the SDK-86, each in two ROMs: a keyboard monitor, for use with the kit's on-board keypad, and a serial monitor that supports use with a CRT or TTY connected to the serial interface (J_7 on the SDK-86). Because the keypad does not really prove tremendously efficient for development, a logical first step is to set up the system for serial-monitor use.

Power-up and reset operations on the kit cause the 8086 to begin execution of whatever program is at location FF000H; this is the usual location of the keyboard-monitor ROM. To enter the serial monitor, you can either execute a GO command to location FE000H from the keyboard (the location of the serial-monitor ROM) or simply swap ROM sets on the board—a preferable alternative because then when you initialize or reset the system you will already be in the serial monitor. The only other action required to complete serial-monitor entry is to connect your CRT or TTY to the board's serial interface.

MCS-86 Symbol	MCS-80 Symbol	Meaning
AX	None	Accumulator (16-bit) (8080 Accumulator holds only 8 bits)
AH	None	Accumulator (high-order byte)
AL	A	Accumulator (low-order byte)
BX	HL	Register B (16-bit) (8080 register pair HL), which may be split and addressed as two 8-bit registers
BH	H	High-order byte of register B
BL	L	Low-order byte of register B
CX	BC	Register C (16-bit) (8080 register pair BC), which may be split and addressed as two 8-bit registers
CH	B	High-order byte of register C
CL	C	Low-order byte of register C
DX	DE	Register D (16-bit) (8080 register DE) which may be split and addressed as two 8-bit registers
DH	D	High-order byte of register D
DL	E	Low-order byte of register D
SP	SP	Stack pointer (16-bit)
BP	None	Base pointer
IP	PC	Instruction pointer (8080 program counter)
Flags	Flags	16-bit register space, in which nine flags reside. (Not directly equivalent to 8080 PSW, which contains five flags and the contents of the accumulator)
DI	None	Data index register
SI	None	Stack index register
CS	None	Data segment register
DS	None	Data segment register
ES	None	Extra segment register
SS	None	Stack segment register
REG8		The name of an 8-bit CPU register location
REG16		The name of a 16-bit CPU register location
reg		In the description of an instruction, a field which defines REG8 or REG16
EA		Effective address (16-bit)
r/m		A register name or memory address in an instruction, this 3-bit field defines EA in conjunction with the mode and w fields
mode		In an instruction, this 2-bit field defines addressing mode
w		A 1-bit field in an instruction, identifying byte instructions (w=0), and word instructions (w=1)
d		A 1-bit field identifying direction,j ie, which location is source and which is destination, in an instruction
(...)		Parentheses enclosing a register name or the contents of register or memory location...
(BX)		Represents the contents of register BX, which could be used as the address where an 8-bit operand might be located
((BX))		Means this 8-bit operand, the contents of the memory location pointed at by the contents of register BX
(BX) + 1, (BX)		Is the address of a 16-bit operand whose low-order 8 bits reside in the memory location pointed at by the contents of register (BX) + and whose high-order 8 bits reside in the next sequential memory location, BX + 1
((BX) + 1, (BX))		Is the 16-bit operand that resides there
		Field, eg, (AL).low-nibble describes the low-nibble (least significant 4 bits) of the contents of register AL
. .		Concatenation, eg, ((DX) + 1: (DX)) is a 16-bit word which is the concatenation of two 8-bit bytes, the low-order byte in the memory location pointed at by DX and the high-order byte in the next sequential memory location
addr		Address (16-bit) of a byte in memory
addr-low		Least significant byte of an address
addr-high		Most significant byte of an address
addr + 1: addr		Addresses of two consecutive bytes in memory, beginning at addr
data		Immediate operand (8-bit if w=0; 16-bit if w=1)
data-low		Least significant byte of 16-bit data word
data high		Most significant byte of 16-bit data word
disp		Displacement
disp-low		Least significant byte of 16-bit displacement
disp-high		Most significant byte of 16-bit displacement
<==		Assignment
+		Addition
-		Subtraction
*		Multiplication
/		Division
%		Modulo
&		AND
|		Inclusive OR
!!		Exclusive OR

Table 1—The terminology *used in describing the 8086 instruction set is similar to that for the 8080, but there are also some key differences.*

The monitor provides you with some powerful tools for starting

The serial monitor provides a few simple tools that will get you started; the 10 individual commands that it executes appear in **Table 2.** A particularly noteworthy feature is the monitor's single-step execution of routines. Until you build up your confidence in coding for the 8086, you will find it advantageous to use this feature each time you execute a new routine. By single-stepping, you can check the contents of the registers, change registers, check or change memory contents and return to the appropriate instruction or to any address in the program after each step.

As an alternative to this single-step execution, you can use the GO command, which passes control to the program at the location you specify and also lets you specify the memory locations of breakpoints. When a program reaches a breakpoint, control returns to the monitor while the machine's status is preserved; any program step at the breakpoint address is restored. Thus, you can execute a program in segments, bracketing with breakpoints any bugs that might be present.

A problem with language

Although the monitor provides some powerful routines, it isn't a complete solution to develop-

```
; INITIALIZE P1A PORT FOR INPUT
;
        B900F0          MOV CX,00F0H
        8EC1            MOV ES,CX
;
; SET EXTRA-SEGMENT POINTER TO TOP SEGMENT
;
        BAFFFF          MOV DX,0FFFFH
;
; SET DX REGISTER TO CONTROL PORT ADDRESS
;
        B09B            MOV AL,09BH
;
; SET AL REGISTER TO INPUT CONTROL SETTING
;
      26EE              OUT [ES]
;
; OUTPUT AL TO DX IN SEGMENT ES
;
```

Fig 4—Almost identical to the initialization for output, *this routine passes a 9BH in the accumulator to condition the port to accept an input.*

```
; INPUT BYTE FROM INPUT PORT P1A
; INITIALIZED BY THE ROUTINE IN FIG 4
; BYTE RETURNED IN AL
;
        B900F0          MOV CX,0F000H
        8EC1            MOV ES,CX
;
; SET EXTRA-SEGMENT POINTER
;
        BAF9FF          MOV DX,0FFF9H
;
; POINT DX REGISTER TO DATA PORT ADDRESS
;
      26EC              IN [ES]
;
; INPUT TO AL FROM DX IN SEGMENT ES
```

Fig 5—Data entering through the parallel port *is passed to the program in the low-order byte of the accumulator.*

COMMAND	FUNCTION/SYNTAX
S (Substitute Memory)	Displays/modifies memory locations S[W]<addr>,[[<new contents>],]*<cr>
X (Examine/Modify Register)	Displays/modifies 8086 registers X[<reg>][[<new contents>],]*<cr>
D (Display Memory)	Displays block of memory data D[W][,<end addr>]<cr>
M(Move)	Moves block of memory data M<start addr>,<end addr>,<destination addr><cr>
I(Port Input)	Accepts and displays data at input port I[W]<port addr>,[,]*<cr>
O (Port Output)	Outputs data to output port O[W]<port addr>,<data>[,<data>]*<cr>
G (Go)	Transfer 8086 control from monitor to user program G[<start addr>][,<break addr>]<cr>
N (Single Step)	Executes single user program instruction N[<start addr>],[[<start addr>],]*<cr>
R (Read Hex File)	Reads hexadecimal object file from paper tape into memory R[<bias number>]<cr>
W (Write Hex File)	Outputs block of memory data to paper tape punch W[X]<start addr>,<end addr>[,<exec addr>]<cr>

Table 2—Use the monitor's 10 commands *to aid in making the computer do useful work.*

ment problems. Its listings are provided in PL/M—an expected provision because Intel used that language to write the monitor. To effectively use the monitor routines, though, you must know more about them than you can determine from such high-level-language listings. For starters, you need the addresses of individual routines in order to call them from a program; without such access to the routines (impossible or at least difficult to obtain without an intermediate assembler listing or a link map), you must write your own. After all, you don't want to have to return to the monitor from the middle of a program to manually output data to a port or perform some other equally basic function.

It's necessary, therefore, to construct routines to provide the same functions that the 10 monitor instructions provide. Think of each routine as a building block for future programs, then evaluate each monitor instruction to determine what routines it contains that you might also need. Because we required some of these routines to write this article, part of your work is done, and you can examine our work to understand our development approach.

Input and output

The monitor-output command outputs a character to a port—a requirement for many types of jobs. Thus, because of its high frequency of use in systems, we judged it an excellent place to start creating software tools for the SDK-86.

Joining forces with Robert Grappel of Hemenway Associates Inc, Boston, MA, we attempted to analyze the strategies that the SDK-86's monitor must use to accomplish its I/O. The 8086 addresses I/O ports in the same way that it addresses memory—a feature that indicated to us that the workhorse of our I/O routines should be a MOV command. The analysis worked; the code listed in the accompanying **figures** is the result. A word of caution, however: The assembler code in these listings is only an approximation of the actual 8086 assembler code—a hand-coded pseudodisassembler rendition.

```
;  INITIALIZE SERIAL PORT
;  STATUS IS AT FFF2 IN TOP SEGMENT
;  PORT IMPLEMENTED WITH 8251 USART
;  DATA IS AT FFF0 IN TOP SEGMENT
;
       B900F0          MOV CX,0F000H
       8EC1            MOV ES,CX
;
;  SET EXTRA-SEGMENT REGISTER
;
       BAF2FF          MOV DX,0FFF2H
;
;  POINT DX REGISTER TO STATUS
;
       B065            MOV AL,065H
;
;  BEGIN INITIALIZATION SEQUENCE
;
     26EE              OUT [ES]        ;RESET USART
       B025            MOV AL,025H
     26EE              OUT [ES]        ;CMND
       B065            MOV AL,065H
     26EE              OUT [ES]        ;DTR OFF
       B0CF            MOV AL,0CFH
     26EE              OUT [ES]        ;SET MODE
       B025            MOV AL,025H
     26EE              OUT [ES]        ;CMND
```

Fig 6—The serial port *requires one initialization routine for both input and output.*

Early warning

If you try to use the MCS-86 Assembly Language Reference Manual (9800640) to produce machine code, be aware of a few bugs we found and realize that there could be more:

- JNP is shown encoded as 6BH; it should be 7BH (pg 6-81)
- JP is shown encoded as 72H; it should be 7AH (pg 6-85)
- JA, JNBE and JBE are all shown as 76H. JA and JNBE should be 77H; JBE, 76H.

With these corrections, the assembly manual at least seems to agree with the MCS-86 User's Manual (July 78).

To elaborate, our procedure called for defining the required operations in terms of assembly-language instructions, hand assembling them (looking up the encoding in the manual and trying it) and, when the machine code was fully debugged, translating it back to assembly code. This procedure was necessary because the available tools allowed us only two options: do all the work strictly in machine code, which because of a lack of mnemonics makes explanations tedious and coding awkward, or use the hand-assembly method, which allows planning of coding in assembler terms. In other words, the second option let us think in assembly language—a distinct advantage.

Parallel I/O

Our routines are divided into seven separate entities: four for parallel I/O, three for serial. First take a look at the parallel-I/O capabilities.

Fig 2 illustrates a method of initializing the 8255 parallel-I/O-port circuit for output (power-up and reset operations condition the port for input). The first step in this routine requires two instructions. Why? There's no instruction that allows you to load the ES register immediate; thus the CX register serves as temporary storage for the displacement value to be added to the output port's relative address to produce the actual address.

The routine loads the DX register with the relative (intrasegment) address of the output-port control register. Note that the 8086 reverses the two bytes of the address just like the 8080 (and other μPs) does. Loading the hex value 80

You have to recreate monitor routines for program use

into the low-order byte of the accumulator (AL), and then outputting it to the port's control register, conditions the port to accept data for output.

To output a character from the accumulator (**Fig 3**), you first load the ES register as before, point the DX register to the proper data port (P1A is at FFF9H) and output the contents of AL to the address in DX in the segment stored in ES.

The displacement concept used here is a little confusing at first. RAM is located in the lower segment of memory; I/O is in the upper segment. Sixteen segments of memory (each 64k) are available, and you can store the displacement value in the ES, SS, DS or CS segment register and then use what's termed an override to output data to the proper address and segment. The 26 that precedes the output command EE indicates that the ES register contains the displacement value in our program. **Fig 3b** illustrates the hardware interpretation of this command, and in **Fig 3c** you see how all four segment registers could be used, each pointing to a different segment of memory. For now, however, it's sufficient to get the data to the right address in the correct segment; we'll worry about adding sophistication when more memory is available.

Inputting data from the parallel port requires almost the same code. There are only two differences: First, we now output the hex value 9B to the control register, which conditions the port to accept input (**Fig 4**). Second, the data received by the port is returned in the low-order byte of the accumulator (**Fig 5**).

Serial I/O

Serial I/O requires a slightly different approach than the parallel ports. **Fig 6** presents the only initialization routine necessary; although the strategy behind this routine is the same as before, note that five individual control values are required. This initialization routine is not essential, though, because power-up and reset initialize the serial port for you.

With initialization accomplished, you can execute the I/O routines shown in **Figs 7** and 8. For both output and input you must read the status register. The first bit of the status byte is TXRDY, which indicates that the USART (8251) is ready to output data. The second bit is RXRDY, which indicates that the USART has received data. The routines mask off the appropriate bit (bit 1 for output, bit 2 for input) and AND it with a ONE in the corresponding bit position. When the result indicates not ready

```
;OUTPUT A BYTE TO A SERIAL PORT
;BYTE PASSED IN AL REGISTER
;
        50              PUSH AX
;
;SAVE ACCUMULATOR
;
        B900F0          MOV CX,0F000H
        8EC1            MOV ES,CX
;
;SET EXTRA-SEGMENT REGISTER
;
        BAF2FF          MOV DX,0FFF2H
;
;SET DX TO STATUS
;
      26EC          LOOP: IN [ES]
;
;READ STATUS REGISTER
;
      2401              AND AL,01H
;
;CHECK TXRDY BIT IN STATUS
;
      74FA              JE LOOP
;
;WAIT FOR TXRDY
;
      58                POP AX
;
;RECOVER ACCUMULATOR
;
        BAF0FF          MOV DX,0FFF0H
;
;SET DX REGISTER TO DATA
;
      26EE              OUT [ES]
;
;OUTPUT AL TO SERIAL PORT
;
```

Fig 7—Before data can output *from the serial port, this routine must know that the TRANSMIT READY (TXRDY) bit is set.*

```
;INPUT A BYTE FROM SERIAL PORT
;BYTE RETURNED IN AL REGISTER
;
        B900F0          MOV CX,0F000H
        8EC1            MOV ES,CX
;
;SET EXTRA-SEGMENT REGISTER
;
      . BAF2FF          MOV DX,0FFF2H
;
;SET DX REGISTER TO STATUS
;
      26EC          LOOP: IN [ES]
;
;READ STATUS REGISTER
;
      2402              AND AL,02H
;
;WAIT FOR RXRDY BIT
;
        74FA            JE LOOP
;
        BAF0FF          MOV DX,0FFF0H
;
;SET DX REGISTER TO DATA
;
      26EC              IN [ES]
;
;GET DATA CHARACTER
```

Fig 8—This serial-data input routine *loops until the RECEIVE READY (RXRDY) bit indicates that the USART has received data.*

REGISTER NAME	ABBREVIATION
ACCUMULATOR	AX
BASE	BX
COUNT	CX
DATA	DX
STACK POINTER	SP
BASE POINTER	BP
STACK INDEX	SI
DESTINATION INDEX	DI
CODE SEGMENT	CS
DATA SEGMENT	DS
STACK SEGMENT	SS
EXTRA SEGMENT	ES
INSTRUCTION POINTER	IP
FLAG	FL

Table 3—The 8086 provides 14 registers, *four of which are segment registers that can point to any of the SDK-86's 16 64k segments.*

(ZERO) each routine loops and reads the status bit again until the USART is ready.

In the output routine, the top word on the stack is then moved into the accumulator by the POP AX—restoring the contents of the accumulator, which contains the character you want to output. This step is naturally not required for the input routine.

By analyzing the routines outlined in **Figs 2** through 8, you can begin to learn how to get started programming the 8086; one of the tricks is understanding the μP's addressing schemes. Our byte-oriented routines, however, are only one way of accomplishing the desired tasks; to take full advantage of the 16-bit μP you will want to rewrite them for word I/O. This procedure involves using a second parallel port, but that's no problem because the SDK-86 provides three in each of its two 8255s. **EDN**

A macro assembler eases the task of 8086 cross-assembler writing

System development without good tools is a formidable task; you can use existing hardware and software to make the job easier.

Jack Hemenway and **Edward Teja,**
Associate Editors

Soon after you fully acquaint yourself with the potential of the 8086, you find that you really need more sophisticated tools than a few routines written in object code and a monitor program can provide. If you've followed our suggestion (EDN, January 20, pgs 81-88) and worked on producing machine-code I/O routines for this μP, you should now be ready to abandon machine code in favor of an assembler. We can even provide an additional reason for switching to an assembler: The experience of writing such a system tool gives you a grasp of your processor's instruction set that you really can't obtain any other way.

Macro assembler produces cross assembler

If you have a lot of patience, enough memory on your 8086 board and suitable peripherals, you could conceivably write an assembler for the 8086 from scratch. You would be forced to write it in machine code, however, and that is exactly what you want to avoid.

Fortunately, there is another alternative. Most engineers today have access to some kind of computer; be it a mini, micro or mainframe, it provides the solution to the dilemma. In essence, you use a macro assembler resident on this existing computer to write another assembler—one that accepts inputs in the computer's assembly code and produces 8086-object-code outputs. This arrangement is termed a cross assembler.

The development system we use at EDN comprises a Southwest Technical Products Corp 6800 CPU with 32k of RAM, dual Icom floppy discs, a Lear Siegler ADM-3A CRT terminal and a Centronics printer. There's no magic to this particular set of components, although there is a significant advantage in using a disc-based system capable of producing hard copy.

We developed our cross assembler with Hemenway Associates Inc's RA6800ML macro assembler (modified to produce a condensed output-listing format and to eliminate conflicts between macro names and 6800 opcodes), as well as the firm's EDIT68 text editor. Note the lesson here: The key is to use software that is already available.

Like every other approach to assembler development, this one presents a compromise, and we'll air it before we go any further. Specifically, the resulting cross assembler's syntax is not identical to 8086 assembly-language syntax. It's close, but there are limitations to the capabilities of a cross assembler written with a macro assembler. This problem isn't as bad as it sounds, however, because there already are syntactical differences among the resident assemblers for each processor. Thus, using a cross assembler adds one more element to the confusion but doesn't *create* the confusion.

The advantages of macros

With these facts in mind, we still elected to use a macro assembler to generate our cross assembler—for a very good reason. To see why, first examine the nature of macros.

A macro facility in a language provides extensibility; ie, it permits you to add new statements to the language by defining how those statements will be translated into statements of the original language. For example, a teacher can introduce new words to a child by translating them into other, more familiar words. To explain the word vilify, for instance, the teacher would point out that vile is the same as very bad, and that to vilify something is to speak very badly of it.

Using the same technique, you can cause an existing assembler with a macro facility to recognize the syntax of a new assembly language. Each statement type of the new assembler is thus an extension to the existing assembler, produced by defining it with a macro.

The 8086, for example, has an instruction called TRANSLATE (XLAT), which performs a

Macros allow production of a straightforward cross assembler

table-lookup byte translation. It uses the AL register as an index into a 256-byte table addressed by the BX register; the addressed byte operand transfers to the AL register. No equivalent instruction exists on our 6800 development system's CPU. The required cross assembler, therefore, must possess a definition for this instruction. For the 6800 assembler, the required macro is

```
XLAT    MACRO
        FCB $D7
        MEND
```

With this macro, whenever the assembler encounters instruction XLAT, it assembles a D7 (hex) 1-byte constant; this D7 is the 8086 opcode for the XLAT instruction. The macro definition (called a prototype) can contain any legal assembler or processor instruction except another macro definition.

What macro features are required?

Ideally, the macro features of the assembler you use to construct a cross assembler should include

- Multicharacter macro names
- Multiargument macros
- Macro nesting
- Conditional assembly (IFC...NIFC)
- SET pseudo-op.

The value of multicharacter macro names is that you can use them to more readily approximate the desired syntax. And the multiargument feature simplifies coding. These two features are fairly straightforward; a more subtle one is the use of macro nesting—the definition of one macro in terms of another. During the process of expanding a macro, an assembler with this capability can encounter another macro name in the mnemonic field. Each time it encounters such a macro call within a macro expansion, the complete state of the current macro is placed on a push-down stack, and expansion of the new definition commences. At the end of the nested macro's expansion, the outer macro's saved state is restored (pulled from the push-down stack). The number of macros you can nest in this manner is determined by the size of the push-down stack.

The fourth necessary macro feature involves the use of a conditional-assembly statement—one which is interpreted and executed during the macro-expansion process and which permits the selection and reordering of statements during the macro expansion. This feature allows use of the same macro for similar—but not identical—tasks. For example, you might want the result of a sequence of arithmetic instructions left in a particular register in one case and stored somewhere else in another. **Fig 1** illustrates the effect of such a conditional assembly. In essence, the conditional-assembly facility allows you to

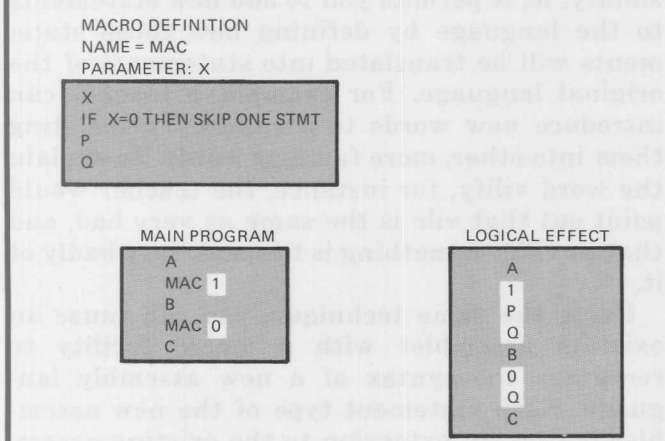


Fig 1—Conditional assembly *lets you use one macro to perform more than one job.*

(a)

```
        ; INITIALIZE P1A PORT FOR INPUT
        ;
              B900F0          MOV CX,00F0H
              8EC1            MOV ES,CX
        ;
        ; SET EXTRA-SEGMENT POINTER TO TOP SEGMENT
        ;
              BAFFFF          MOV DX,0FFFFH
        ;
        ; SET DX REGISTER TO CONTROL PORT ADDRESS
        ;
              B09B            MOV AL,09BH
        ;
        ; SET AL REGISTER TO INPUT CONTROL SETTING
        ;
           26EE               OUT [ES]
        ;
        ; OUTPUT AL TO DX IN SEGMENT ES
        ;
```

(b)

```
0018                    *
0019                    * INITIALIZATION FOR PARALLEL INPUT
0020                    *
0021                    INITIP MOVWTRI  CX,IOSEG
0022 +  0000 B9
0028 +  0001 00
0029 +  0002 F0
0030                    *
0031                           MOVTSG   ES
0032 +  0003 8E
0034                           REGISTER CX
0035 +  0004 C1
0045                    *
0046                           MOVWTRI  DX,PCNTL
0047 +  0005 BA
0053 +  0006 FF
0054 +  0007 FF
0055                    *
0056                           MOVBTRI  AL,$9B
0057 +  0008 B0
0058 +  0009 9B
0059                    *
0060                           SEGOVR   ES
0061 +  000A 26
0062                           OUTB
0068 +  000B EE
0070                    *
```

Fig 2—Hand-assembled code (a) *written for a previous article initializes a parallel port for input. The cross-assembler output* **(b)** *is identical to it. The plus sign after some line numbers indicates that the code results from a macro expansion.*

control which statements are assembled in response to various forms of macro calls and passed parameters.

The final required macro feature—the SET pseudo-op—assigns to a symbol a value other than the value normally assigned to it by the program-location counter. The SET statement contains a number, symbol or expression in its operand field and functions like the EQU pseudo-op, except that the symbols may be defined more than once. The current value of the label in the symbol table is always the value assignment from the last SET statement. You need the SET statement if you wish to use a macro with given symbols more than once—during each expansion of a macro its symbols can be assigned appropriate values.

The foregoing five macro functions provide you with the power to create a cross assembler without having to program in machine code. The availability and usefulness of an aid like the 6800 text editor are themselves sufficient to make this approach worthy of consideration.

Instructions divide into classes

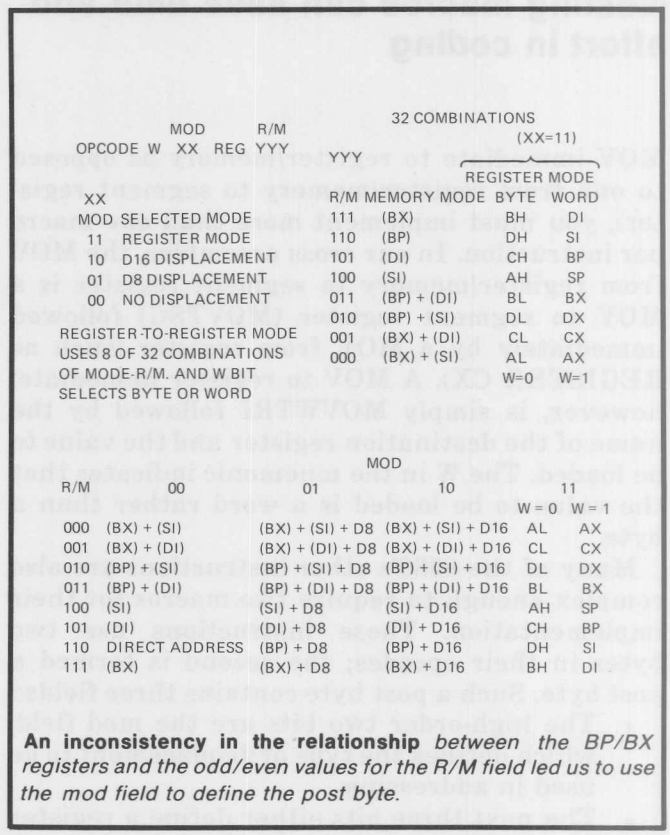
You might find it convenient or even necessary (depending on the power of your macro assembler) to break down the target processor's instruction set into as many as three classes of instructions. The first class contains those instructions—like XLAT—that can be implemented with a single macro definition. The second class comprises those that require separate macros to deal with particular instruction fields, as you'll see later when we discuss the MOV instructions. The third class includes instructions that share certain sequences of code. In these cases, rather than rewrite a section of code for each macro, you can use the nesting principle to ease the job—in much the same way that you use common subroutines.

For example, consider the byte reversal of addresses in the 8086. Because the 6800 does not byte-reverse addresses, it's necessary to code for it. The macro that accomplishes this task is

```
REVRS   MACRO
        .DW &1
        FCB .D2
        FCB .D1
        MEND
```

This isn't a particularly complex sequence of code, but with it at your disposal, when writing the macro to move a word from the accumulator you need only write

```
MOVWFA  MACRO
        FCB $A3
        .REVRS &1
        MEND
```



An inconsistency in the relationship *between the BP/BX registers and the odd/even values for the R/M field led us to use the mod field to define the post byte.*

rather than

```
MOVWFA  MACRO
        FCB $A3
        .DW &1
        FCB .D2
        FCB .D1
        MEND
```

As a bonus, the .REVRS macro is available for use in other macros. Note that .DW is itself a macro used by .REVRS; it sets .D1 and .D2 to the high and low bytes of its argument. Thus, if you had to write out all the code for MOVWFA, it would read

```
MOVWFA  MACRO
        FCB $A3
.D1     SET &1/256
.D2     SET .D1*256
.D2     SET &1-.D2
        FCB .D2
        FCB .D1
        MEND
```

The final product takes shape

The 8086 cross assembler we constructed according to the foregoing principles requires 246 individual macro prototypes—a feature arising partly because of the 8086's extensive instruction set and partly because of some "thrashing about" required to get the desired output.

As noted, the MOV instructions represent one such class of problems. The 8086 allows 32 addressing modes; to distinguish one type of MOV command from another (for example, a

Nesting macros can save time and effort in coding

MOV immediate to register/memory as opposed to one from register/memory to segment register), you must implement more than one macro per instruction. In our cross assembler, the MOV from register/memory to segment register is a MOV to segment register (MOVTSG) followed immediately by a MOV from register (such as REGISTER CX). A MOV to register immediate, however, is simply MOVWTRI followed by the name of the destination register and the value to be loaded. The W in the mnemonic indicates that the value to be loaded is a word rather than a byte.

Many of the 8086's other instructions are also complex enough to require two macros for their implementation. These instructions use two bytes in their opcodes; the second is termed a post byte. Such a post byte contains three fields:

- The high-order two bits are the mod field, which defines the type of displacement to be used in addressing.
- The next three bits either define a register or represent a fixed pattern.
- The low-order three bits constitute the R/M (register/memory) field, which defines the base and index registers used for addressing.

Encoding a post byte presents the most difficult aspect of writing our macro-assembler-based cross assembler. The mechanism we used allows the instruction macro to pass the middle field of the post byte to a second (post-byte) macro that completes the post-byte definition. The field is passed in a symbol-table entry termed .XXX.

A careful look at the 8086 addressing modes shown in the nearby **table** suggests that the mod field should define the set of post-byte macros. The primary reason for using this field rather than the R/M field lies in an inconsistency in the encoding of the latter. Specifically, the odd/even relationship of the BP and BX registers is not maintained for all cases. Hence, the direct-address (ABSOLUTE) case requires special and quite complex handling.

There are five types of post-byte macros. REGISTER, defines mod=3 and fills R/M with the register argument of the macro. ABSOLUTE defines mod=0 and R/M=6 (a special case that uses no base or index registers, only an immediate 16-bit address). NODISP defines mod=0 with the R/M field set by the argument passed to the macro. Finally, LONG and SHORT define mod=2 and mod=1, respectively.

The 8086 allows eight possible combinations of the two base registers (BX and BP) and two index registers (SI and DI). Special symbols in the macro set handle all eight of these combinations. BXSI, for example, implies that the BX register is the base and SI is the index. Alternatively, the no-displacement form of addressing with BX as the base pointer and DI as the index register requires coding the post-byte macro as

NODISP BXDI

The assembler converts this coding to a hex byte with mod=0 and R/M=1, with the middle field set by the instruction macro.

As a second example, consider how to achieve addressing with the BP base, word displacement and no index. This form requires coding

LONG BPNI,DISP

In a similar manner, SHORT provides byte displacement.

As a simple test of our cross assembler, we

Using the post-byte macros

As explained in the text, five post-byte macros allow implementation of the more complex 8086 instructions—those that are too complex to be handled by a single macro. The following macros involve the use of one of the five post-byte macros.

ADCBRF
ADCBI
ADCBSI
ADCBTR
ADCWFR
ADCWI
ADCWSI
ADCWTR
ADDBFR
ADDBI
ADDBSI
ADDBTR
ADDWFR
ADDWI
ADDWSI
ADDWTR
ANDBFR
ANDBI
ANDBTR
ANDWFR
ANDWI
ANDWTR
CALLI
CALLIS
CMPBFR
CMPBI
CMPBSI
CMPBTR
CMPWFR
CMPWI
CMPWSI
CMPWTR
DECB
DECW
DIVB
DIVW
IDIVB
IDIVW
IMULB
IMULW
INCB
INCW
JMPI
JMPIS
LDS
LEA
LES
MOVBI
MOVBMR
MOVBRM
MOVFSG
MOVTSG
MOVWI
MOVWMR
MOVWRM
MULB
MULW
NEGB
NEGW
NOTB
NOTW
ORBFR
ORBI
ORBTR
ORWFR
ORWI
ORWTR
POP
PUSH
RCLB
RCLW
RCRB
RCRW
ROLB
ROLW
RORB
RORW
SALB
SALW
SARB
SARW
SBBBFR
SBBBI
SBBBSI
SBBBTR
SBBWFR
SBBWI
SBBWSI
SSBWTR
SHLB
SHLW
SHRB
SHRW
SUBBFR
SUBBI
SUBBSI
SUBBTR
SUBWFR
SUBWI
SUBWSI
SUBWTR
TESTBI
TESTBR
TESTWI
TESTWR
XCHGB
XCHGW
XORBFR
XORBI
XORBTR
XORWFR
XORWI
XORWTR

```
0001  * INTEL 8086 CROSS-ASSEMBLER
0002  * MACRO-SET IMPLEMENTATION
0003  *
0004  * COPYRIGHT 1978 BY HEMENWAY ASSOCIATES INC.
0005  * BOSTON MASS.  ALL RIGHTS RESERVED
0006  * WRITTEN BY ROBERT D.  GRAPPEL
0007  *
0008  AX      EQU 0      16-BIT REGISTER DEFINITIONS
0009  CX      EQU 1
0010  DX      EQU 2
0011  BX      EQU 3
0012  SP      EQU 4
0013  BP      EQU 5
0014  SI      EQU 6
0015  DI      EQU 7
0016  *
0017  AL      EQU 0      8-BIT REGISTER DEFINITIONS
0018  CL      EQU 1
0019  DL      EQU 2
0020  BL      EQU 3
0021  AH      EQU 4
0022  CH      EQU 5
0023  DH      EQU 6
0024  BH      EQU 7
0025  *
0026  BXSI    EQU 0      R/M FIELD DEFINITIONS
0027  BXDI    EQU 1
0028  BPSI    EQU 2
0029  BPDI    EQU 3
0030  NBSI    EQU 4
0031  NBDI    EQU 5
0032  BPNI    EQU 6
0033  BXNI    EQU 7
0034  *
0035  ES      EQU 0      SEGMENT-REGISTER DEFINITIONS
0036  CS      EQU 1
0037  SS      EQU 2
0038  DS      EQU 3
0039  *
0040  *
0041  * DATA-DEFINITION PSEUDO-OPERATIONS
0042  *
0043  DB      MACRO
0044  * DEFINE BYTE
0045          FCB &1
0046          MEND
0047  *
0048  DW      MACRO
0049  * DEFINE WORD
0050          .REVRS &1
0051          MEND
0052  *
0053  * ADDRESSING-TYPE MACROS
0054  *
0055  ABSOLUTE MACRO
0056          FCB .XXX+6  ASSUME .XXX SET BY INSTR.MACRO
0057          .REVRS &1   OUTPUT DISP-LOW, DISP-HIGH
0058          .DATA
0059          MEND
0060  *
0061  REGISTER MACRO
0062          FCB $C0+.XXX+&1
0063          .DATA
0064          MEND
0065  *
0066  LONG    MACRO
0067  * 16-BIT DISPLACEMENT
0068          FCB $80+.XXX+&1
0069          .DW &2
0070          FCB .D1        OUTPUT DISP-HIGH, DISP-LOW
0071          FCB .D2
0072          .DATA
0073          MEND
0074  *
0075  SHORT   MACRO
0076  * 8-BIT DISPLACEMENT
0077          FCB $40+.XXX+&1
0078          FCB &2
0079          .DATA
0080          MEND
0081  *
0082  NODISP MACRO
0083  * NO DISPLACEMENT
0084          FCB .XXX+&1
0085          .DATA
0086          MEND
0087  *
0088  .DW     MACRO
0089  *
0090  * SET .D1 TO HIGH BYTE OF ARG
0091  * SET .D2 TO LOW BYTE OF ARG
0092  *
0093  .D1     SET &1/256
0094  .D2     SET .D1*256
0095  .D2     SET &1-.D2
0096          MEND
0097  *
0098  .REVRS MACRO
0099  *
0100  * FORM DISPLACEMENT BYTE-REVERSED
0101  *
0102          .DW &1
0103          FCB .D2
0104          FCB .D1
0105          MEND
0106  *
0107  .DATA   MACRO
0108  * FORM DATA FIELD OF IMMEDIATE-REGISTER/MEMORY INSTS.
0109          IFC .BYTE
0110  * BYTE OF IMM.  DATA NEEDED?
0111          FCB .IDAT
0112  .BYTE   SET 0
0113          NIFC
0114  *
0115          IFC .WORD
0116  * WORD OF IMM.  DATA NEEDED?
0117          .REVRS .IDAT
0118  .WORD   SET 0
0119          NIFC
0120          MEND
0121  *
0122  .BYTE   SET 0
0123  .WORD   SET 0
0124  *
0125  *
0126  * INSTRUCTION-DEFINITION MACROS
0127  *
0128  AAA     MACRO
0129  * ASCII ADJUST FOR ADDITION
0130          FCB $37
0131          MEND
0132  *
0133  AAD     MACRO
0134  * ASCII ADJUST FOR DIVISION
0135          FDB $D50A
0136          MEND
0137  *
0138  AAM     MACRO
0139  * ASCII ADJUST FOR MULTIPLICATION
0140          FDB $C40A
0141          MEND
0142  *
0143  AAS     MACRO
0144  * ASCII ADJUST FOR SUBTRACTION
0145          FCB $3F
0146          MEND
0147  *
0148  ADCBA   MACRO
0149  * ADD BYTE TO ACCUMULATOR IMMEDIATE WITH CARRY
0150          FCB $14
0151          FCB &1
0152          MEND
0153  *
0154  ADCBFR MACRO
0155  * ADD REGISTER BYTE WITH CARRY, SOURCE= REGISTER
0156          FCB $10
0157  .XXX    SET &1*8
0158          MEND
0159  *
0160  ADCBI   MACRO
0161  * ADD UNSIGNED BYTE WITH CARRY IMMEDIATE
0162          FCB $80
0163  .XXX    SET $10
0164  .BYTE   SET 1
0165  .IDAT   SET &1
0166          MEND
0167  *
0168  ADCBSI MACRO
0169  * ADD SIGNED BYTE WITH CARRY IMMEDIATE
0170          FCB $82
0171  .XXX    SET $10
0172  .BYTE   SET 1
0173  .IDAT   SET &1
0174          MEND
0175  *
0176  ADCBTR MACRO
0177  * ADD REGISTER BYTE WITH CARRY, SOURCE=ADDRESS
0178          FCB $12
0179  .XXX    SET &1*8
0180          MEND
0181  *
0182  ADCWA   MACRO
0183  * ADD WORD TO ACCUMULATOR IMMEDIATE WITH CARRY
0184          FCB $15
0185           REVRS &1
0186          MEND
```

Fig 3—We needed 246 individual macros *to implement a complete cross assembler for the 8086 on our 6800 system. Note the special use of &0 to represent the number of arguments in the macro call.*

```
0187 *
0188 ADCWFR MACRO
0189 * ADD REGISTER WORD WITH CARRY, SOURCE=REGISTER
0190         FCB $11
0191 .XXX    SET &1*8
0192         MEND
0193 *
0194 ADCWI   MACRO
0195 * ADD UNSIGNED WORD WITH CARRY IMMEDIATE
0196         FCB $81
0197 .XXX    SET $10
0198 .WORD   SET 1
0199 .IDAT   SET &1
0200         MEND
0201 *
0202 ADCWSI MACRO
0203 * ADD SIGNED WORD WITH CARRY IMMEDIATE
0204         FCB $83
0205 .XXX    SET $10
0206 .WORD   SET 1
0207 .IDAT   SET &1
0208         MEND
0209 *
0210 ADCWTR MACRO
0211 * ADD REGISTER WORD WITH CARRY, SOURCE=ADDRESS
0212         FCB $13
0213 .XXX    SET &1*8
0214         MEND
0215 *
0216 ADDBA   MACRO
0217 * ADD BYTE TO ACCUMULATOR IMMEDIATE
0218         FCB $04
0219         FCB &1
0220         MEND
0221 *
0222 ADDBFR MACRO
0223 * ADD REGISTER BYTE, SOURCE=REGISTER
0224         FCB $00
0225 .XXX    SET &1*8
0226         MEND
0227 *
0228 ADDBI   MACRO
0229 * ADD UNSIGNED BYTE IMMEDIATE
0230         FCB $80
0231 .XXX    SET 0
0232 .BYTE   SET 1
0233 .IDAT   SET &1
0234         MEND
0235 *
0236 ADDBSI MACRO
0237 * ADD SIGNED BYTE IMMEDIATE
0238         FCB $82
0239 .XXX    SET 0
0240 .BYTE   SET 1
0241 .IDAT   SET &1
0242         MEND
0243 *
0244  ADDBTR MACRO
0245  * ADD REGISTER BYTE, SOURCE=ADDRESS
0246          FCB $02
0247 .XXX     SET &1*8
0248          MEND
0249 *
0250  ADDWA   MACRO
0251  * ADD WORD TO ACCUMULATOR IMMEDIATE
0252          FCB $05
0253          .REVRS &1
0254          MEND
0255 *
0256  ADDWFR MACRO
0257  * ADD REGISTER WORD, SOURCE=REGISTER
0258          FCB $01
0259 .XXX     SET &1*8
0260          MEND
0261 *
0262  ADDWI   MACRO
0263  * ADD UNSIGNED WORD IMMEDIATE
0264          FCB $81
0265 .XXX     SET 0
0266 .WORD    SET 1
0267 .IDAT    SET &1
0268          MEND
0269 *
0270 ADDWSI MACRO
0271 * ADD SIGNED WORD IMMEDIATE
0272         FCB $83
0273 .XXX    SET 0
0274 .WORD   SET 1
0275 .IDAT   SET &1
0276         MEND
0277 *
0278 ADDWTR MACRO
0279 * ADD REGISTER WORD, SOURCE=ADDRESS
0280         FCB $03
0281 .XXX    SET &1*8
0282         MEND
0283 *
0284 ANDBA   MACRO
0285 * AND BYTE WITH ACCUMULATOR IMMEDIATE
0286         FCB $24
0287         FCB &1
0288         MEND
0289 *
0290 ANDBFR MACRO
0291 * AND REGISTER BYTE, SOURCE=REGISTER
0292         FCB $20
0293 .XXX    SET &1*8
0294         MEND
0295 *
0296 ANDBI   MACRO
0297 * AND IMMEDIATE BYTE WITH REGISTER/MEMORY
0298         FCB $80
0299 .XXX    SET $20
0300 .BYTE   SET 1
0301 .IDAT   SET &1
0302         MEND
0303 *
0304 ANDBTR MACRO
0305 * AND REGISTER BYTE, SOURCE=ADDRESS
0306         FCB $22
0307 .XXX    SET &1*8
0308         MEND
0309 *
0310 ANDWA   MACRO
0311 * AND WORD WITH ACCUMULATOR IMMEDIATE
0312         FCB $25
0313         .REVRS &1
0314         MEND
0315 *
0316 ANDWFR MACRO
0317 * AND REGISTER WORD, SOURCE=REGISTER
0318         FCB $21
0319 .XXX    SET &1*8
0320         MEND
0321 *
0322 ANDWI   MACRO
0323 * AND IMMEDIATE WORD WITH REGISTER/MEMORY
0324         FCB $81
0325 .XXX    SET $20
0326 .WORD   SET 1
0327 .IDAT   SET &1
0328         MEND
0329 *
0330 ANDWTR MACRO
0331 * AND REGISTER WORD, SOURCE=ADDRESS
0332         FCB $23
0333 .XXX    SET &1*8
0334         MEND
0335 *
0336 CALLD   MACRO
0337 * CALL DIRECT WITHIN SEGMENT  (ARG=DISP)
0338         FCB $E8
0339 .LOC    SET *+2
0340         .REVRS &1-.LOC
0341         MEND
0342 *
0343 CALLDS MACRO
0344 * CALL DIRECT INTERSEGMENT (ARG1=DISP, ARG2=SEGMENT)
0345         FCB $9A
0346 .LOC    SET *+4
0347         .REVRS &1-.LOC
0348         .REVRS &2
0349         MEND
0350 *
0351 CALLI   MACRO
0352 * CALL INDIRECT WITHIN SEGMENT
0353         FCB $FF
0354 .XXX    SET $10
0355         MEND
0356 *
0357 CALLIS MACRO
0358 * CALL INDIRECT INTERSEGMENT
0359         FCB $FF
0360 .XXX    SET $18
0361         MEND
0362 *
0363 CBW     MACRO
0364 * CONVERT WORD TO BYTE
0365         FCB $98
0366         MEND
0367 *
0368 CLC     MACRO
0369 * CLEAR CARRY FLAG
0370         FCB $F8
0371         MEND
0372 *
0373 CLD     MACRO
0374 * CLEAR DIRECTION FLAG
0375         FCB $FC
0376         MEND
0377 *
0378 CLI     MACRO
0379 * CLEAR INTERRUPT FLAG
0380         FCB $FA
0381         MEND
0382 *
```

```
0383 CMC     MACRO
0384 * COMPLEMENT CARRY FLAG
0385         FCB $F5
0386         MEND
0387 *
0388 CMPBA   MACRO
0389 * COMPARE BYTE WITH ACCUMULATOR IMMEDIATE
0390         FCB $3C
0391         FCB &1
0392         MEND
0393 *
0394 CMPBFR MACRO
0395 * COMPARE REGISTER BYTE, SOURCE=REGISTER
0396         FCB $38
0397 . XXX    SET &1*8
0398         MEND
0399 *
0400 CMPBI   MACRO
0401 * COMPARE UNSIGNED BYTE IMMEDIATE
0402         FCB $80
0403 . XXX    SET $38
0404 . BYTE   SET 1
0405 . IDAT   SET &1
0406         MEND
0407 *
0408 CMPBSI MACRO
0409 * COMPARE SIGNED BYTE IMMEDIATE
0410         FCB $82
0411 . XXX    SET $38
0412 . BYTE   SET 1
0413 . IDAT   SET &1
0414         MEND
0415 *
0416 CMPBTR MACRO
0417 * COMPARE REGISTER BYTE, SOURCE=ADDRESS
0418         FCB $3A
0419 . XXX    SET &1*8
0420         MEND
0421 *
0422 CMPSB   MACRO
0423 * COMPARE STRING BYTE
0424         FCB $A6
0425         MEND
0426 *
0427 CMPSW   MACRO
0428 * COMPARE STRING WORD
0429         FCB $A7
0430         MEND
0431 *
0432 CMPWFR MACRO
0433 * COMPARE REGISTER WORD, SOURCE=REGISTER
0434         FCB $39
0435 . XXX    SET &1*8
0436         MEND
0437 *
0438 CMPWI   MACRO
0439 * COMPARE UNSIGNED WORD IMMEDIATE
0440         FCB $81
0441 . XXX    SET $38
0442 . WORD   SET 1
0443 . IDAT   SET &1
0444         MEND
0445 *
0446 CMPWSI MACRO
0447 * COMPARE SIGNED WORD IMMEDIATE
0448         FCB $83
0449 . XXX    SET $38
0450 . WORD   SET 1
0451 . IDAT   SET &1
0452         MEND
0453 *
0454 CMPWTR MACRO
0455 * COMPARE REGISTER WORD, SOURCE=ADDRESS
0456         FCB $3B
0457 . XXX    SET &1*8
0458         MEND
0459 *
0460 DAA     MACRO
0461 * DECIMAL ADJUST FOR ADDITION
0462         FCB $27
0463         MEND
0464 *
0465 DAS     MACRO
0466 * DECIMAL ADJUST FOR SUBTRACTION
0467         FCB $2F
0468         MEND
0469 *
0470 DECB    MACRO
0471 * DECREMENT BYTE
0472 . XXX    SET $08
0473         FCB $FE
0474         MEND
0475 *
0476 DECR    MACRO
0477 * DECREMENT REGISTER
0478         FCB $48+&1
0479         MEND
0480 *
0481 DECW    MACRO
0482 * DECREMENT WORD
0483 . XXX    SET $08
0484         FCB $FF
0485         MEND
0486 *
0487 DIVB    MACRO
0488 * DIVIDE BYTE
0489 . XXX    SET $30
0490         FCB $F6
0491         MEND
0492 *
0493 DIVW    MACRO
0494 * DIVIDE WORD
0495 . XXX    SET $30
0496         FCB $F7
0497         MEND
0498 *
0499 ESC     MACRO
0500 * ESCAPE (ADDRESS OUTPUT)
0501 . XXX    SET 0
0502         FCB $D8
0503         MEND
0504 *
0505 HLT     MACRO
0506 * HALT PROCESSOR
0507         FCB $F4
0508         MEND
0509 *
0510 IDIVB   MACRO
0511 * INTEGER DIVIDE BYTE (SIGNED)
0512 . XXX    SET $38
0513         FCB $F6
0514         MEND
0515 *
0516 IDIVW   MACRO
0517 * INTEGER DIVIDE WORD (SIGNED)
0518 . XXX    SET $38
0519         FCB $F7
0520         MEND
0521 *
0522 IMULB   MACRO
0523 * INTEGER MULTIPLY BYTE (SIGNED)
0524 . XXX    SET $28
0525         FCB $F6
0526         MEND
0527 *
0528 IMULW   MACRO
0529 * INTEGER MULTIPLY WORD (SIGNED)
0530 . XXX    SET $28
0531         FCB $F7
0532         MEND
0533 *
0534 INB     MACRO
0535         IFC &0
0536 * FIXED-PORT BYTE INPUT
0537         FCB $E4
0538         FCB &1
0539         NIFC
0540 *
0541         IFC &0-1
0542 * VARIABLE-PORT BYTE INPUT
0543         FCB $EC
0544         NIFC
0545         MEND
0546 *
0547 INCB    MACRO
0548 * INCREMENT BYTE
0549 . XXX    SET $00
0550         FCB $FE
0551         MEND
0552 *
0553 INCR    MACRO
0554 * INCREMENT REGISTER
0555         FCB $40+&1
0556         MEND
0557 *
0558 INCW    MACRO
0559 * INCREMENT WORD
0560 . XXX    SET $00
0561         FCB $FF
0562         MEND
0563 *
0564 INT     MACRO
0565 * INTERRUPT PROCESS
0566         FCB $CD
0567         FCB &1
0568         MEND
0569 *
0570 INT3    MACRO
0571 * TYPE "3" INTERRUPT
0572         FCB $CC
0573         MEND
0574 *
0575 INTO    MACRO
0576 * INTERRUPT ON OVERFLOW
0577         FCB $CE
0578         MEND
```

```
0579 *
0580 INW     MACRO
0581         IFC &0
0582 * FIXED-PORT WORD INPUT
0583         FCB $E5
0584         FCB &1
0585         NIFC
0586 *
0587         IFC &0-1
0588 * VARIABLE-PORT WORD INPUT
0589         FCB $ED
0590         NIFC
0591         MEND
0592 *
0593 IRET    MACRO
0594 * INTERRUPT RETURN
0595         FCB $CF
0596         MEND
0597 *
0598 .OFFST  MACRO
0599 *
0600 * FORM 8-BIT OFFSET FOR JUMPS,LOOPS
0601 *
0602 .LOC    SET *+1
0603         FCB &1-.LOC
0604 *
0605         MEND
0606 *
0607 JA      MACRO
0608 * JUMP IF ABOVE
0609         FCB $77
0610         .OFFST &1
0611         MEND
0612 *
0613 JAE     MACRO
0614 * JUMP IF ABOVE OR EQUAL
0615         FCB $73
0616         .OFFST &1
0617         MEND
0618 *
0619 JB      MACRO
0620 * JUMP IF BELOW
0621         FCB $72
0622         .OFFST &1
0623         MEND
0624 *
0625 JBE     MACRO
0626 * JUMP IF BELOW OR EQUAL
0627         FCB $76
0628         .OFFST &1
0629         MEND
0630 *
0631 JCXZ    MACRO
0632 * JUMP IF CX REGISTER=ZERO
0633         FCB $E3
0634         .OFFST &1
0635         MEND
0636 *
0637 JE      MACRO
0638 * JUMP IF EQUAL
0639         FCB $74
0640         .OFFST &1
0641         MEND
0642 *
0643 JG      MACRO
0644 * JUMP IF GREATER THAN
0645         FCB $7F
0646         .OFFST &1
0647         MEND
0648 *
0649 JGE     MACRO
0650 * JUMP IF GREATER THAN OR EQUAL TO
0651         FCB $7D
0652         .OFFST &1
0653         MEND
0654 *
0655 JL      MACRO
0656 * JUMP IF LESS THAN
0657         FCB $7C
0658         .OFFST &1
0659         MEND
0660 *
0661 JLE     MACRO
0662 * JUMP IF LESS THAN OR EQUAL TO
0663         FCB $7E
0664         .OFFST &1
0665         MEND
0666 *
0667 JMPD    MACRO
0668 * JUMP DIRECT WITHIN SEGMENT (ARG=DISP)
0669         FCB $E9
0670 .LOC    SET *+2
0671         .REVRS &1-.LOC
0672         MEND
0673 *
0674 JMPDS   MACRO
0675 * JUMP DIRECT INTERSEGMENT (ARG1=DISP, ARG2=SEGMENT)
0676         FCB $EA
0677 .LOC    SET *+4
0678         .REVRS &1-.LOC
0679         .REVRS &2
0680         MEND
0681 *
0682 JMPI    MACRO
0683 * JUMP INDIRECT WITHIN SEGMENT
0684         FCB $FF
0685 .XXX    SET $20
0686         MEND
0687 *
0688 JMPIS   MACRO
0689 * JUMP INDIRECT INTERSEGMENT
0690         FCB $FF
0691 .XXX    SET $28
0692         MEND
0693 *
0694 JMPS    MACRO
0695 * JUMP DIRECT WITHIN SEGMENT SHORT (ARG=DISP-SHORT)
0696         FCB $EB
0697         .OFFST &1
0698         MEND
0699 *
0700 JNA     MACRO
0701 * JUMP IF NOT ABOVE
0702         JBE &1
0703         MEND
0704 *
0705 JNAE    MACRO
0706 * JUMP IF NOT ABOVE OR EQUAL
0707         JB &1
0708         MEND
0709 *
0710 JNB     MACRO
0711 * JUMP IF NOT BELOW
0712         JAE &1
0713         MEND
0714 *
0715 JNBE    MACRO
0716 * JUMP IF NOT BELOW OR EQUAL
0717         FCB $77
0718         .OFFST &1
0719         MEND
0720 *
0721 JNE     MACRO
0722 * JUMP IF NOT EQUAL
0723         FCB $75
0724         .OFFST &1
0725         MEND
0726 *
0727 JNG     MACRO
0728 * JUMP IF NOT GREATER THAN
0729         JLE &1
0730         MEND
0731 *
0732 JNGE    MACRO
0733 * JUMP IF NOT GREATER THAN OR EQUAL TO
0734         JL &1
0735         MEND
0736 *
0737 JNL     MACRO
0738 * JUMP IF NOT LESS THAN
0739         JGE &1
0740         MEND
0741 *
0742 JNLE    MACRO
0743 * JUMP IF NOT LESS THAN OR EQUAL TO
0744         JG &1
0745         MEND
0746 *
0747 JNO     MACRO
0748 * JUMP IF NO OVERFLOW
0749         FCB &71
0750         .OFFST &1
0751         MEND
0752 *
0753 JNP     MACRO
0754 * JUMP IF NO PARITY
0755         FCB $7B
0756         .OFFST &1
0757         MEND
0758 *
0759 JNS     MACRO
0760 * JUMP IF NOT SIGN FLAG
0761         FCB $79
0762         .OFFST &1
0763         MEND
0764 *
0765 JNZ     MACRO
0766 * JUMP IF NOT ZERO
0767         JNE &1
0768         MEND
0769 *
0770 JO      MACRO
0771 * JUMP IF OVERFLOW
0772         FCB $70
0773         .OFFST &1
0774         MEND
```

```
0775 *
0776 JP      MACRO
0777 * JUMP IF PARITY
0778         FCB $7A
0779         .OFFST &1
0780         MEND
0781 *
0782 JPE     MACRO
0783 * JUMP IF EVEN PARITY
0784         JP &1
0785         MEND
0786 *
0787 JPO     MACRO
0788 * JUMP IF ODD PARITY
0789         JNP &1
0790         MEND
0791 *
0792 JS      MACRO
0793 * JUMP IF SIGN FLAG
0794         FCB $78
0795         .OFFST &1
0796         MEND
0797 *
0798 JZ      MACRO
0799 * JUMP IF ZERO
0800         JE &1
0801         MEND
0802 *
0803 LAHF    MACRO
0804 * LOAD AH REGISTER WITH FLAGS
0805         FCB $9F
0806         MEND
0807 *
0808 LDS     MACRO
0809 * LOAD POINTER INTO DS
0810 .XXX    SET &1*8
0811         FCB $C5
0812         MEND
0813 *
0814 LEA     MACRO
0815 * LOAD EFFECTIVE ADDRESS
0816 .XXX    SET &1*8
0817         FCB $8D
0818         MEND
0819 *
0820 LES     MACRO
0821 * LOAD POINTER INTO ES
0822 .XXX    SET &1*8
0823         FCB $C4
0824         MEND
0825 *
0826 LOCK    MACRO
0827 * LOCK BUS PREFIX
0828         FCB $F0
0829         MEND
0830 *
0831 LODSB   MACRO
0832 * LOAD BYTE OF STRING
0833         FCB $AC
0834         MEND
0835 *
0836 LODSW   MACRO
0837 * LOAD WORD OF STRING
0838         FCB $AD
0839         MEND
0840 *
0841 LOOP    MACRO
0842 * LOOP ON CX REGISTER
0843         FCB $E2
0844         .OFFST &1
0845         MEND
0846 *
0847 LOOPE   MACRO
0848 * LOOP WHILE EQUAL
0849         FCB $E1
0850         .OFFST &1
0851         MEND
0852 *
0853 LOOPNE  MACRO
0854 * LOOP WHILE NOT EQUAL
0855         FCB $E0
0856         .OFFST &1
0857         MEND
0858 *
0859 LOOPNZ  MACRO
0860 * LOOP WHILE NOT ZERO
0861         LOOPNE &1
0862         MEND
0863 *
0864 LOOPZ   MACRO
0865 * LOOP WHILE ZERO
0866         LOOPE &1
0867         MEND
0868 *
0869 MOVBFA  MACRO
0870 * MOVE BYTE FROM ACCUMULATOR
0871         FCB $A2
0872         .REVRS &1
0873         MEND
0874 *
0875 MOVBI   MACRO
0876 * MOVE IMMEDIATE BYTE
0877         FCB $C6
0878 .XXX    SET 0
0879 .BYTE   SET 1
0880 .IDAT   SET &1
0881         MEND
0882 *
0883 MOVBMR  MACRO
0884 * MOVE BYTE TO REGISTER
0885         FCB $8A
0886 .XXX    SET &1*8
0887         MEND
0888 *
0889 MOVBRM  MACRO
0890 * MOVE BYTE FROM REGISTER
0891         FCB $88
0892 .XXX    SET &1*8
0893         MEND
0894 *
0895 MOVBTA  MACRO
0896 * MOVE BYTE TO ACCUMULATOR
0897         FCB $A0
0898         .REVRS &1
0899         MEND
0900 *
0901 MOVBTRI MACRO
0902 * MOVE BYTE TO REGISTER IMMEDIATE
0903         FCB $B0+&1
0904         FCB &2
0905         MEND
0906 *
0907 MOVFSG  MACRO
0908 * MOVE FROM SEGMENT REGISTER
0909         FCB $8C
0910 .XXX    SET &1*8
0911         MEND
0912 *
0913 MOVSB   MACRO
0914 * MOVE BYTE OF STRING
0915         FCB $A4
0916         MEND
0917 *
0918 MOVSW   MACRO
0919 * MOVE WORD OF STRING
0920         FCB $A5
0921         MEND
0922 *
0923 MOVTSG  MACRO
0924 * MOVE TO SEGMENT REGISTER
0925         FCB $8E
0926 .XXX    SET &1*8
0927         MEND
0928 *
0929 MOVWFA  MACRO
0930 * MOVE WORD FROM ACCUMULATOR
0931         FCB $A3
0932         .REVRS &1
0933         MEND
0934 *
0935 MOVWI   MACRO
0936 * MOVE WORD IMMEDIATE
0937         FCB $C7
0938 .XXX    SET 0
0939 .WORD   SET 1
0940 .IDAT   SET &1
0941         MEND
0942 *
0943 MOVWMR  MACRO
0944 * MOVE WORD TO REGISTER
0945         FCB $8B
0946 .XXX    SET &1*8
0947         MEND
0948 *
0949 MOVWRM  MACRO
0950 * MOVE WORD FROM REGISTER
0951         FCB $89
0952 .XXX    SET &1*8
0953         MEND
0954 *
0955 MOVWTA  MACRO
0956 * MOVE WORD TO ACCUMULATOR
0957         FCB $A1
0958         .REVRS &1
0959         MEND
0960 *
0961 MOVWTRI MACRO
0962 * MOVE WORD TO REGISTER IMMEDIATE
0963         FCB $B8+&1
0964         .REVRS &2
0965         MEND
0966 *
0967 MULB    MACRO
0968 * MULTIPLY BYTE
0969 .XXX    SET $20
0970         FCB $F6
```

```
0971          MEND
0972  *
0973  MULW    MACRO
0974  * MULTIPLY WORD
0975   XXX    SET $20
0976          FCB $F7
0977          MEND
0978  *
0979  NEGB    MACRO
0980  * NEGATE BYTE
0981  . XXX   SET $18
0982          FCB $F6
0983          MEND
0984  *
0985  NEGW    MACRO
0986  * NEGATE WORD
0987  . XXX   SET $18
0988          FCB $F7
0989          MEND
0990  *
0991  *
0992  NOP     MACRO
0993  * NO OPERATION
0994          FCB $90
0995          MEND
0996  *
0997  NOTB    MACRO
0998  * COMPLEMENT BYTE
0999  . XXX   SET $10
1000          FCB $F6
1001          MEND
1002  *
1003  NOTW    MACRO
1004  * COMPLEMENT WORD
1005  . XXX   SET $10
1006          FCB $F7
1007          MEND
1008  *
1009  ORBA    MACRO
1010  * OR ACCUMULATOR BYTE IMMEDIATE
1011          FCB $0C
1012          FCB &1
1013          MEND
1014  *
1015  ORBFR   MACRO
1016  * OR REGISTER BYTE, SOURCE=REGISTER
1017          FCB $08
1018  . XXX   SET &1*8
1019          MEND
1020  *
1021  ORBI    MACRO
1022  * OR IMMEDIATE BYTE WITH REGISTER/MEMORY
1023          FCB $80
1024  . XXX   SET $08
1025  . BYTE  SET 1
1026  . IDAT  SET &1
1027          MEND
1028  *
1029  ORBTR   MACRO
1030  * OR REGISTER BYTE, SOURCE=ADDRESS
1031          FCB $0A
1032  . XXX   SET &1*8
1033          MEND
1034  *
1035  ORWA    MACRO
1036  * OR ACCUMULATOR WORD IMMEDIATE
1037          FCB $0D
1038          . REVRS &1
1039          MEND
1040  *
1041  ORWFR   MACRO
1042  * OR REGISTER WORD, SOURCE=REGISTER
1043          FCB $09
1044  . XXX   SET &1*8
1045          MEND
1046  *
1047  ORWI    MACRO
1048  * OR IMMEDIATE WORD WITH REGISTER/MEMORY
1049          FCB $81
1050  . XXX   SET $08
1051  . WORD  SET 1
1052  . IDAT  SET &1
1053          MEND
1054  *
1055  ORWTR   MACRO
1056  * OR REGISTER WORD, SOURCE=ADDRESS
1057          FCB $0B
1058  . XXX   SET &1*8
1059          MEND
1060  *
1061  OUTB    MACRO
1062          IFC &0
1063  * FIXED-PORT BYTE OUTPUT
1064          FCB $E6
1065          FCB &1
1066          NIFC
1067  *
1068          IFC &0-1
1069  * VARIABLE-PORT BYTE OUTPUT
1070          FCB $EE
1071          NIFC
1072          MEND
1073  *
1074  OUTW    MACRO
1075          IFC &0
1076  * FIXED-PORT WORD OUTPUT
1077          FCB $E7
1078          FCB &1
1079          NIFC
1080  *
1081          IFC &0-1
1082  * VARIABLE-PORT WORD OUTPUT
1083          FCB $EF
1084          NIFC
1085          MEND
1086  *
1087  POP     MACRO
1088  * POP WORD FROM STACK
1089  . XXX   SET $00
1090          FCB $8F
1091          MEND
1092  *
1093  POPF    MACRO
1094  * POP FLAGS FROM STACK
1095          FCB $9D
1096          MEND
1097  *
1098  POPR    MACRO
1099  * POP REGISTER FROM STACK
1100          FCB $58+&1
1101          MEND
1102  *
1103  POPSR   MACRO
1104  * POP SEGMENT REGISTER FROM STACK
1105          FCB &1*8+$07
1106          MEND
1107  *
1108  PUSH    MACRO
1109  * PUSH WORD ONTO STACK
1110  . XXX   SET $30
1111          FCB $FF
1112          MEND
1113  *
1114  PUSHF   MACRO
1115  * PUSH FLAGS ONTO STACK
1116          FCB $9C
1117          MEND
1118  *
1119  PUSHR   MACRO
1120  * PUSH REGISTER ONTO STACK
1121          FCB $50+&1
1122          MEND
1123  *
1124  PUSHSR  MACRO
1125  * PUSH SEGMENT REGISTER ONTO STACK
1126          FCB &1*8+$06
1127          MEND
1128  *
1129  RCLB    MACRO
1130  * ROTATE BYTE LEFT THROUGH CARRY
1131  . XXX   SET $10
1132  . YYY   SET $D0
1133          IFC &0
1134  . YYY   SET $D2
1135          NIFC
1136          FCB . YYY
1137          MEND
1138  *
1139  RCLW    MACRO
1140  * ROTATE WORD LEFT THROUGH CARRY
1141  . XXX   SET $10
1142  . YYY   SET $D1
1143          IFC &0
1144  . YYY   SET $D3
1145          NIFC
1146          FCB . YYY
1147          MEND
1148  *
1149  RCRB    MACRO
1150  * ROTATE BYTE RIGHT THROUGH CARRY
1151  . XXX   SET $18
1152  . YYY   SET $D0
1153          IFC &0
1154  . YYY   SET $D2
1155          NIFC
1156          FCB . YYY
1157          MEND
1158  *
1159  RCRW    MACRO
1160  * ROTATE WORD RIGHT THROUGH CARRY
1161  . XXX   SET $18
1162  . YYY   SET $D1
1163          IFC &0
1164  . YYY   SET $D3
1165          NIFC
1166          FCB . YYY
```

```
1167          MEND
1168  *
1169  REP     MACRO
1170  * REPEAT UNTIL CX REGISTER=ZERO
1171          FCB $F2
1172          MEND
1173  *
1174  REPE    MACRO
1175  * REPEAT WHILE EQUAL
1176          FCB $F2
1177          MEND
1178  *
1179  REPNE   MACRO
1180  * REPEAT WHILE NOT EQUAL
1181          FCB $F3
1182          MEND
1183  *
1184  REPNZ   MACRO
1185  * REPEAT WHILE NOT ZERO
1186          FCB $F2
1187          MEND
1188  *
1189  REPZ    MACRO
1190  * REPEAT WHILE ZERO
1191          FCB $F3
1192          MEND
1193  *
1194  RET     MACRO
1195  * RETURN WITHIN SEGMENT
1196          FCB $C3
1197          MEND
1198  *
1199  RETA    MACRO
1200  * RETURN WITHIN SEGMENT WITH IMM.  ADDITION TO SP
1201          FCB $C2
1202          .REVRS &1
1203          MEND
1204  *
1205  RETI    MACRO
1206  * RETURN INTERSEGMENT
1207          FCB $CB
1208          MEND
1209  *
1210  RETIA   MACRO
1211  * RETURN INTERSEGMENT WITH IMM.  ADDITION TO SP
1212          FCB $CA
1213          .REVRS &1
1214          MEND
1215  *
1216  ROLB    MACRO
1217  * ROTATE BYTE LEFT
1218  .XXX    SET $00
1219  .YYY    SET $D0
1220          IFC &0
1221  .YYY    SET $D2
1222          NIFC
1223          FCB .YYY
1224          MEND
1225  *
1226  ROLW    MACRO
1227  * ROTATE WORD LEFT
1228  .XXX    SET $00
1229  .YYY    SET $D1
1230          IFC &0
1231  .YYY    SET $D3
1232          NIFC
1233          FCB .YYY
1234          MEND
1235  *
1236  RORB    MACRO
1237  * ROTATE BYTE RIGHT
1238  .XXX    SET $08
1239  .YYY    SET $D0
1240          IFC &0
1241  .YYY    SET $D2
1242          NIFC
1243          FCB .YYY
1244          MEND
1245  *
1246  RORW    MACRO
1247  * ROTATE WORD RIGHT
1248  .XXX    SET $08
1249  .YYY    SET $D1
1250          IFC &0
1251  .YYY    SET $D3
1252          NIFC
1253          FCB .YYY
1254          MEND
1255  *
1256  SAHF    MACRO
1257  * STORE AH REGISTER INTO FLAGS
1258          FCB $9E
1259          MEND
1260  *
1261  SALB    MACRO
1262  * SHIFT BYTE LEFT ARITHMETIC
1263  .XXX    SET $20
1264  .YYY    SET $D0
1265          IFC &0
1266  .YYY    SET $D2
1267          NIFC
1268          FCB .YYY
1269          MEND
1270  *
1271  SALW    MACRO
1272  * SHIFT WORD LEFT ARITHMETIC
1273  .XXX    SET $20
1274  .YYY    SET $D1
1275          IFC &0
1276  .YYY    SET $D3
1277          NIFC
1278          FCB .YYY
1279          MEND
1280  *
1281  SARB    MACRO
1282  * SHIFT BYTE RIGHT ARITHMETIC
1283  .XXX    SET $38
1284  .YYY    SET $D0
1285          IFC &0
1286  .YYY    SET $D2
1287          NIFC
1288          FCB .YYY
1289          MEND
1290  *
1291  SARW    MACRO
1292  * SHIFT WORD RIGHT ARITHMETIC
1293  .XXX    SET $38
1294  .YYY    SET $D1
1295          IFC &0
1296  .YYY    SET $D3
1297          NIFC
1298          FCB .YYY
1299          MEND
1300  *
1301  SBBBA   MACRO
1302  * SUBTRACT BYTE FROM ACCUMULATOR IMMMEDIATE WITH BORROW
1303          FCB $1C
1304          FCB &1
1305          MEND
1306  *
1307  SBBBFR  MACRO
1308  * SUBTRACT REGISTER BYTE WITH BORROW, SOURCE=REGISTER
1309          FCB $18
1310  .XXX    SET &1*8
1311          MEND
1312  *
1313  SBBBI   MACRO
1314  * SUBTRACT UNSIGNED BYTE WITH BORROW IMMEDIATE
1315          FCB $80
1316  .XXX    SET $18
1317  .BYTE   SET 1
1318  .IDAT   SET &1
1319          MEND
1320  *
1321  SBBBSI  MACRO
1322  * SUBTRACT SIGNED BYTE WITH BORROW IMMEDIATE
1323          FCB $82
1324  .XXX    SET $18
1325  .BYTE   SET 1
1326  .IDAT   SET &1
1327          MEND
1328  *
1329  SBBBTR  MACRO
1330  * SUBTRACT REGISTER BYTE WITH BORROW, SOURCE=ADDRESS
1331          FCB $1A
1332  .XXX    SET &1*8
1333          MEND
1334  *
1335  SBBWA   MACRO
1336  * SUBTRACT WORD FROM ACCUMULATOR IMMEDIATE WITH BORROW
1337          FCB $1D
1338          .REVRS &1
1339          MEND
1340  *
1341  SBBWFR  MACRO
1342  * SUBTRACT REGISTER WORD WITH BORROW, SOURCE=REGISTER
1343          FCB $19
1344  .XXX    SET &1*8
1345          MEND
1346  *
1347  SBBWI   MACRO
1348  * SUBTRACT UNSIGNED WORD WITH BORROW IMMEDIATE
1349          FCB $81
1350  .XXX    SET $18
1351  .WORD   SET 1
1352  .IDAT   SET &1
1353          MEND
1354  *
1355  SBBWSI  MACRO
1356  * SUBTRACT SIGNED WORD WITH BORROW IMMEDIATE
1357          FCB $83
1358  .XXX    SET $18
1359  .WORD   SET 1
1360  .IDAT   SET &1
1361          MEND
1362  *
```

```
1363 SBBWTR MACRO
1364 * SUBTRACT REGISTER WORD WITH BORROW, SOURCE=ADDRESS
1365         FCB $1B
1366 . XXX    SET &1*8
1367         MEND
1368 *
1369 SCASB   MACRO
1370 * SCAN STRING BYTE
1371         FCB $AE
1372         MEND
1373 *
1374 SCASW   MACRO
1375 * SCAN STRING WORD
1376         FCB $AF
1377         MEND
1378 *
1379 SEGOVR MACRO
1380 *
1381 * SEGMENT OVERRIDES
1382 * CS,DS,ES,SS
1383 *
1384         FCB &1*8+$26
1385         MEND
1386 *
1387 SHLB    MACRO
1388 * SHIFT BYTE LEFT
1389 . XXX    SET $20
1390 . YYY    SET $D0
1391         IFC &0
1392 . YYY    SET $D2
1393         NIFC
1394         FCB . YYY
1395         MEND
1396 *
1397 SHLW    MACRO
1398 * SHIFT WORD LEFT
1399 . XXX    SET $20
1400 . YYY    SET $D1
1401         IFC &0
1402 . YYY    SET $D3
1403         NIFC
1404         FCB . YYY
1405         MEND
1406 *
1407 SHRB    MACRO
1408 * SHIFT BYTE RIGHT
1409 . XXX    SET $28
1410 . YYY    SET $D0
1411         IFC &0
1412 . YYY    SET $D2
1413         NIFC
1414         FCB . YYY
1415         MEND
1416 *
1417 SHRW    MACRO
1418 * SHIFT WORD RIGHT
1419 . XXX    SET $28
1420 . YYY    SET $D1
1421         IFC &0
1422 . YYY    SET $D3
1423         NIFC
1424         FCB . YYY
1425         MEND
1426 *
1427 STC     MACRO
1428 * SET CARRY FLAG
1429         FCB $F9
1430         MEND
1431 *
1432 STD     MACRO
1433 * SET DIRECTION FLAG
1434         FCB $FD
1435         MEND
1436 *
1437 STI     MACRO
1438 * SET INTERRUPT FLAG
1439         FCB $FB
1440         MEND
1441 *
1442 STOSB   MACRO
1443 * STORE BYTE OF STRING
1444         FCB $AA
1445         MEND
1446 *
1447 STOSW   MACRO
1448 * STORE WORD OF STRING
1449         FCB $AB
1450         MEND
1451 *
1452 SUBBA   MACRO
1453 * SUBTRACT BYTE FROM ACCUMULATOR IMMEDIATE
1454         FCB $2C
1455         FCB &1
1456         MEND
1457 *
1458 SUBBFR MACRO
1459 * SUBTRACT REGISTER BYTE, SOURCE=REGISTER
1460         FCB $28
1461 . XXX    SET &1*8
1462         MEND
1463 *
1464 SUBBI   MACRO
1465 * SUBTRACT UNSIGNED BYTE IMMEDIATE
1466         FCB $80
1467 . XXX    SET $28
1468 . BYTE   SET 1
1469 . IDAT   SET &1
1470         MEND
1471 *
1472 SUBBSI MACRO
1473 * SUBTRACT SIGNED BYTE IMMEDIATE
1474         FCB $82
1475 . XXX    SET $28
1476 . BYTE   SET 1
1477 . IDAT   SET &1
1478         MEND
1479 *
1480 SUBBTR MACRO
1481 * SUBTRACT REGISTER BYTE, SOURCE=ADDRESS
1482         FCB $2A
1483 . XXX    SET &1*8
1484         MEND
1485 *
1486 SUBWA   MACRO
1487 * SUBTRACT WORD FROM ACCUMULATOR IMMEDIATE
1488         FCB $2D
1489         . REVRS &1
1490         MEND
1491 *
1492 SUBWFR MACRO
1493 * SUBTRACT REGISTER WORD, SOURCE=REGISTER
1494         FCB $29
1495 . XXX    SET &1*8
1496         MEND
1497 *
1498 SUBWI   MACRO
1499 * SUBTRACT UNSIGNED WORD IMMEDIATE
1500         FCB $82
1501 . XXX    SET $28
1502 . WORD   SET 1
1503 . IDAT   SET &1
1504         MEND
1505 *
1506 SUBWSI MACRO
1507 * SUBTRACT SIGNED WORD IMMEDIATE
1508         FCB $83
1509 . XXX    SET $28
1510 . WORD   SET 1
1511 . IDAT   SET &1
1512         MEND
1513 *
1514 SUBWTR MACRO
1515 * SUBTRACT REGISTER WORD, SOURCE=ADDRESS
1516         FCB $2B
1517 . XXX    SET &1*8
1518         MEND
1519 *
1520 TESTBA MACRO
1521 * TEST BYTE IN ACCUMULATOR
1522         FCB $A8
1523         FCB &1
1524         MEND
1525 *
1526 TESTBI MACRO
1527 * TEST IMMEDIATE BYTE WITH REGISTER/MEMORY
1528         FCB $F6
1529 . XXX    SET 0
1530 . BYTE   SET 1
1531 . IDAT   SET &1
1532         MEND
1533 *
1534 TESTBR MACRO
1535 * TEST BYTE IN REGISTER
1536         FCB $84
1537 . XXX    SET &1*8
1538         MEND
1539 *
1540 TESTWA MACRO
1541 * TEST WORD IN ACCUMULATOR
1542         FCB $A9
1543         . REVRS &1
1544         MEND
1545 *
1546 TESTWI MACRO
1547 * TEST IMMEDIATE WORD WITH REGISTER/MEMORY
1548         FCB $F7
1549 . XXX    SET 0
1550 . WORD   SET 1
1551 . IDAT   SET &1
1552         MEND
1553 *
1554 TESTWR MACRO
1555 * TEST WORD IN REGISTER
1556         FCB $85
1557 . XXX    SET &1*8
1558         MEND
```

```
1559  *
1560  WAIT    MACRO
1561  * WAIT
1562          FCB $9B
1563          MEND
1564  *
1565  XCHGB   MACRO
1566  * EXCHANGE BYTE
1567  .XXX    SET &1
1568          FCB $86
1569          MEND
1570  *
1571  XCHGR   MACRO
1572  * EXCHANGE REGISTERS
1573          FCB $90+&1
1574          MEND
1575  *
1576  XCHGW   MACRO
1577  * EXCHANGE WORD
1578  .XXX    SET &1
1579          FCB $87
1580          MEND
1581  *
1582  XORBA   MACRO
1583  * EXCLUSIVE-OR ACCUMULATOR BYTE IMMEDIATE
1584          FCB $34
1585          FCB &1
1586          MEND
1587  *
1588  XORBFR  MACRO
1589  * EXCLUSIVE-OR REGISTER BYTE, SOURCE=REGISTER
1590          FCB $30
1591  .XXX    SET &1*8
1592          MEND
1593  *
1594  XORBI   MACRO
1595  * EXCLUSIVE-OR IMMEDIATE BYTE WITH REGISTER/MEMORY
1596          FCB $80
1597  .XXX    SET $30
1598  .BYTE   SET 1
1599  .IDAT   SET &1
1600          MEND
1601  *
1602  XORBTR  MACRO
1603  * EXCLUSIVE-OR REGISTER BYTE, SOURCE=ADDRESS
1604          FCB $32
1605  .XXX    SET &1*8
1606          MEND
1607  *
1608  XORWA   MACRO
1609  * EXCLUSIVE-OR ACCUMULATOR WORD IMMEDIATE
1610          FCB $35
1611          .REVRS &1
1612          MEND
1613  *
1614  XORWFR  MACRO
1615  * EXCLUSIVE-OR REGISTER WORD, SOURCE=REGISTER
1616          FCB $31
1617  .XXX    SET &1*8
1618          MEND
1619  *
1620  XORWI   MACRO
1621  * EXCLUSIVE-OR IMMEDIATE WORD WITH REGISTER/MEMORY
1622          FCB $81
1623  .XXX    SET $30
1624  .WORD   SET 1
1625  .IDAT   SET &1
1626          MEND
1627  *
1628  XORWTR  MACRO
1629  * EXCLUSIVE-OR REGISTER WORD, SOURCE=ADDRESS
1630          FCB $33
1631  .XXX    SET &1*8
1632          MEND
1633  *
1634  XLAT    MACRO
1635  * TRANSLATE
1636          FCB $D7
1637          MEND
1638  *
1639          END
```

The 8086's 32 addressing modes complicate the MOV instructions

cross-assembled the driver programs that we hand-coded for the January 20 EDN article. These programs provided a good test because we knew the object code that should result from correct assembly of the assembly code. **Fig 2a** shows the hand-assembled code for the initialization of a parallel port for input, while **Fig 2b** shows the equivalent listing from the cross assembler. The machine code that follows each instruction mnemonic results from the appropriate macro expansion. Line 0031, for example, presents the 8086 instruction for the first half of a load to segment register from a register/memory. This procedure produces 8E (hex); the post-byte macro (REGISTER CX) produces C1 (hex).

Fig 3 presents the complete list of macro definitions we used to create our cross assembler. Macros whose names begin with a period can be called by other macros (nested). Naturally, this particular set is useful only with the RA6800ML macro assembler, but if you understand it you can create your own set for whatever macro assembler you have at hand. **EDN**

Adding floppies to an 8086 paves the way for system software

Jack Hemenway and **Edward Teja,**
Associate Editors

After spending some time getting to know the 8086 (EDN, January 20, pg 81, and February 5, pg 115), you will want to do more than wiggle a few bits at an output port. And one piece of armament required to enable a 16-bit μC (or one with any other number of bits) to face the workaday world is a disc system. You could purchase such a system from the μC manufacturer, and there is certainly nothing dishonorable in this plug-in-and-go method. Yet suppose, as is the case at EDN, that a floppy-disc system already sits on the shelf. It makes sense to at least consider putting that system to work.

Configuring the hardware is usually trivial

Whether the floppy system interfaces to an SDK-86 board or an iSBC 86/12 single-board μC, the strategy and wiring remain the same. Our design mates the iSBC 86/12 to a pair of Icom FF36 Frugal Floppies via one 8255A programmable peripheral interface chip (**Fig 1**). If you use an SDK-86, though, its prototyping area provides a convenient place to add the necessary buffers and drivers shown in **Fig 2**. The iSBC board furnishes these buffers and drivers, and so a modified cable serves nicely for the complete interface modification on that μC.

The SDK-86 and iSBC 86/12 use almost identical software for disc routines; the only differences are the locations of the three 8255A ports. The interface chip provides three 8-bit parallel-I/O ports; we designated PA_0 through PA_7 as DATA IN 0 through 7, respectively, PB_0 through PB_7 as COMMAND STROBE and COMMAND WORD and PC_0 through PC_7 as DATA OUT 0 through 7, respectively.

Fig 3 shows the bit definitions for the interface's data, control and status lines. The track definition (**Fig 3a**) shows 77 tracks per diskette, addressed as 0 to 4C hex. Unit/sector assignments (**Fig 3b**) utilize bits 0 through 4 for sector designation and bits 6 and 7 as the unit bits. Bit 5 is always a ZERO. Selection of one of the diskette's 26 sectors occurs through hex addresses 1 through 1A. Data-out (**Fig 3c**) and data-in (**Fig 3d**) bit assignments use all eight bits. Finally, **Fig 3e** furnishes status-bit definition:

- Bit 0=Busy bit
- Bits 1,2=Unit-select code bits. When you select a drive, testing these bits determines if it is the correct one.
- Bit 3=Media or CRC error. Indicates that a READ CRC or READ produced a data error.
- Bit 4=Selected unit write protected
- Bit 5=Drive failure

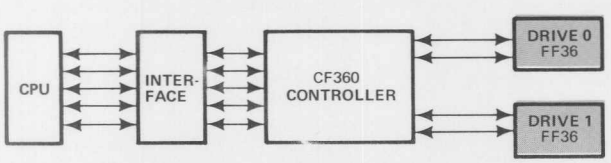


Fig 1—Our floppy-disc configuration *utilizes two 8-in. units from Icom.*

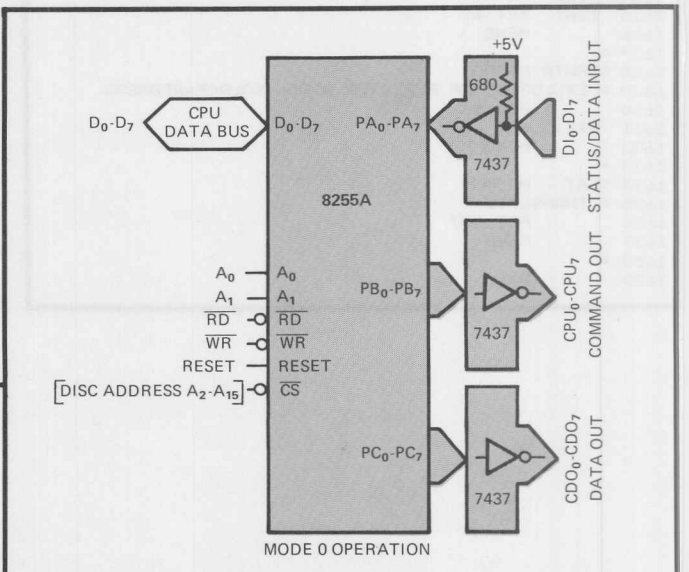


Fig 2—One 8255A controls three 8-bit parallel ports— *just the right number of bits for a complete interface.*

- Bit 6=Not used; always a ONE
- Bit 7=Found Deleted Data Address Mark (DDAM). If the controller finds a DDAM preceding the data during a READ, it sets this status bit. A DDAM is under user control and can flag bad sectors, flag a sector as the last one in a file or flag a sector as unused.

Commanding the disc to work

Fig 4 shows the 14 commands (and their bit patterns) used to effect disc operations. The following points elaborate on these commands:

- EXAMINE STATUS—Places the status bits on the input-data lines
- READ—Reads the contents of a selected unit/sector into the controller's Read buffer
- WRITE—Writes the contents of the controller's Write buffer to a selected unit/sector
- READ CRC—Tests for a CRC error
- SEEK—Steps the head to the desired track
- CLEAR ERROR FLAGS—Clears DDAM and CRC status bits
- SEEK TRACK 0—Steps the head to track 0
- WRITE WITH DDAM—Same as WRITE, with a DDAM preceding the data
- LOAD TRACK ADDRESS—Loads the track address into the controller after the desired track is set up on the data-output lines
- LOAD UNIT/SECTOR—Loads a unit/sector into the controller after that unit/sector is set up on the data-output lines
- LOAD WRITE BUFFER—Transfers data from the data-output lines into the controller's Write buffer
- SHIFT READ BUFFER—Shifts the controller's Read buffer and places a byte from that buffer on the data-input lines
- CLEAR—Halts any operation in progress and clears the Busy status bit
- EXAMINE READ BUFFER—Places the next available byte of the Read buffer's contents on the data-input lines.

Routines make the drives go

GETBUF (**Fig 5**) and WRTBUF (**Fig 6**) perform much of the work in our floppy-disc interface; they read and write full sectors of data to and from a disc. In addition, seven subroutines (**Fig 7**) support these routines.

GETBUF calls XMITUS to transmit the unit/sector information to the controller; it then calls DRIVCK to make sure that the drive is up and running and has responded to that unit/sector information. If not, GETBUF returns with the return code (RC) from DRIVCK; otherwise it directs the drive to seek the desired track. Next, a retry count is initialized to five and a READ command is issued. Upon completion, GETBUF checks for a CRC error and, if one has occurred, the program calls ERFRST to clear the error-status flags, then tries the READ again.

This process repeats up to five times if necessary, then quits after loading an RC of 31_H before returning. If one of the READs is successfully executed, the 128 bytes of data go from the controller's Read buffer to the buffer specified in RAM. The program accomplishes this transfer by

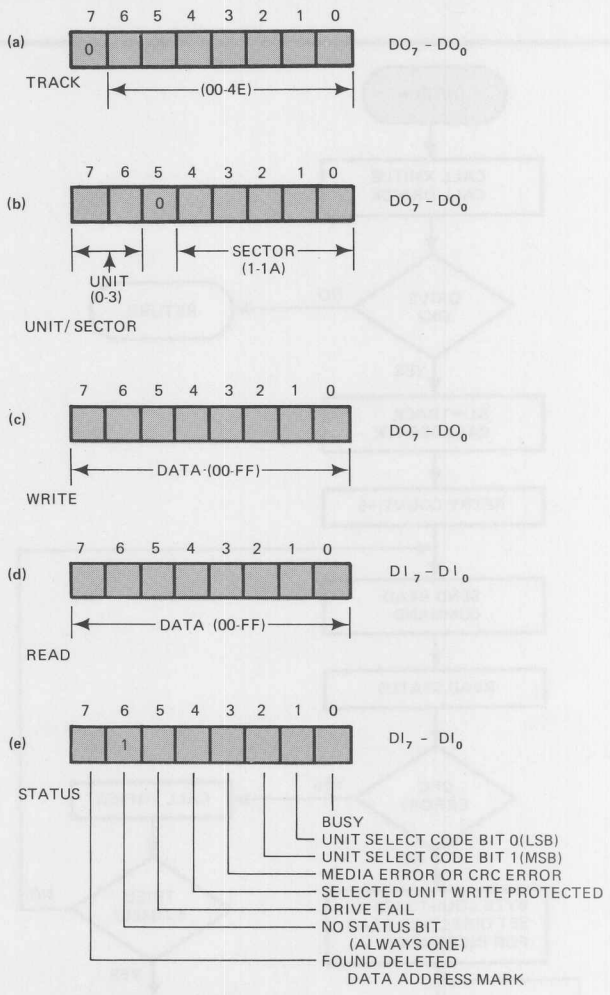


Fig 3—The interface's data/status lines *use eight bits.*

COMMAND	CPU_7-CPU_0	HEX CODE
	76543210	
EXAMINE STATUS	00000000	00
READ	00000011	03
WRITE	00000101	05
READ CRC	00000111	07
SEEK	00001001	09
CLEAR ERROR FLAGS	00001011	0B
SEEK TRACK 0	00001101	0D
WRITE WITH DDAM	00001111	0F
LOAD TRACK ADDRESS	00010001	11
LOAD UNIT/SECTOR	00100001	21
LOAD WRITE BUFFER	00110001	31
SHIFT READ BUFFER	01000001	41
CLEAR	10000001	81
EXAMINE READ BUFFER	01000000	40

Fig 4—An 8-bit command word *provides 14 commands to effect the floppy's operations.*

alternately issuing the SHIFT READ BUFFER command, reading a byte from the input-data lines and storing it. When all of the data has been transferred, the RC is set to 00 and GETBUF returns.

WRTBUF, when called, first saves the unit/sector information, then enters a loop that transfers 128 bytes of data from RAM to the controller's Write buffer. Next, it restores the unit/sector and track information, calls XMITUS to transmit the unit/sector information to the controller and calls DRIVCK to ensure

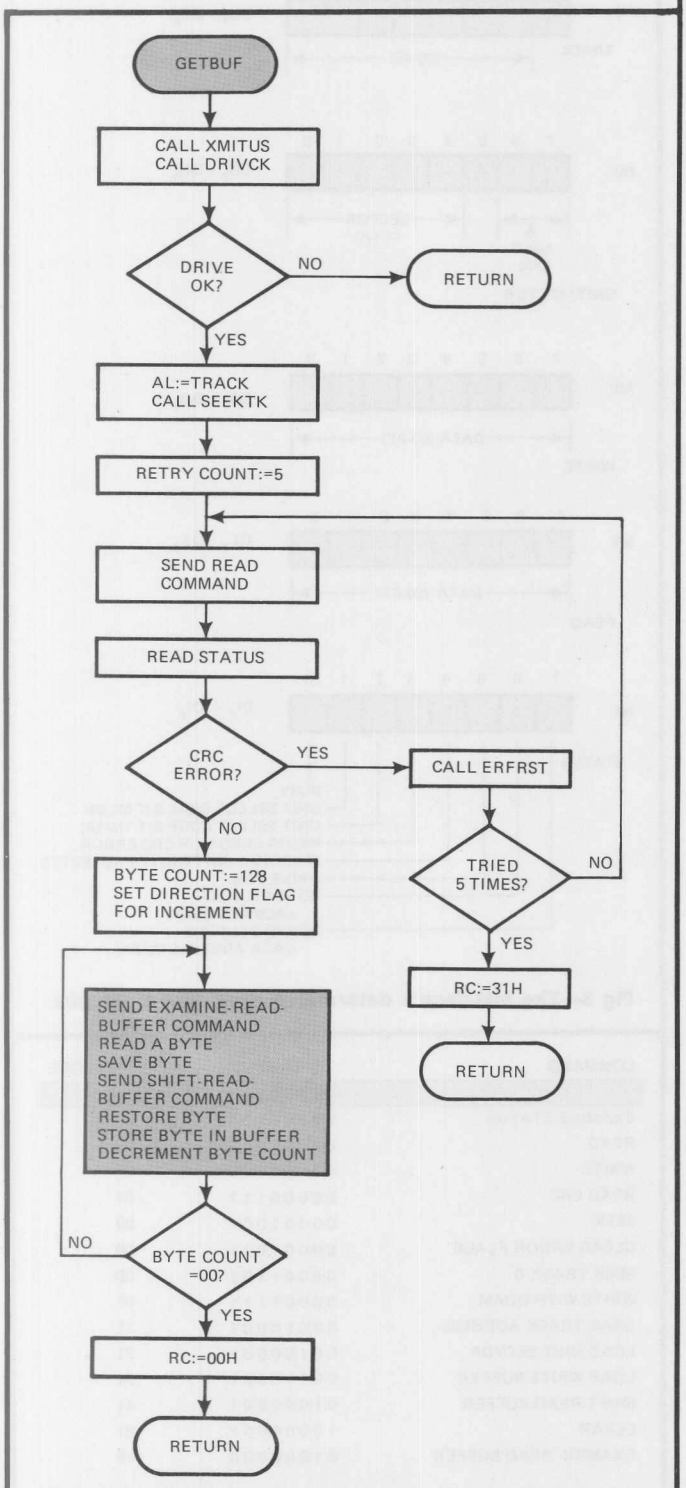


Fig 5—GETBUF reads a full sector from a disc *and stores it in the Read buffer.*

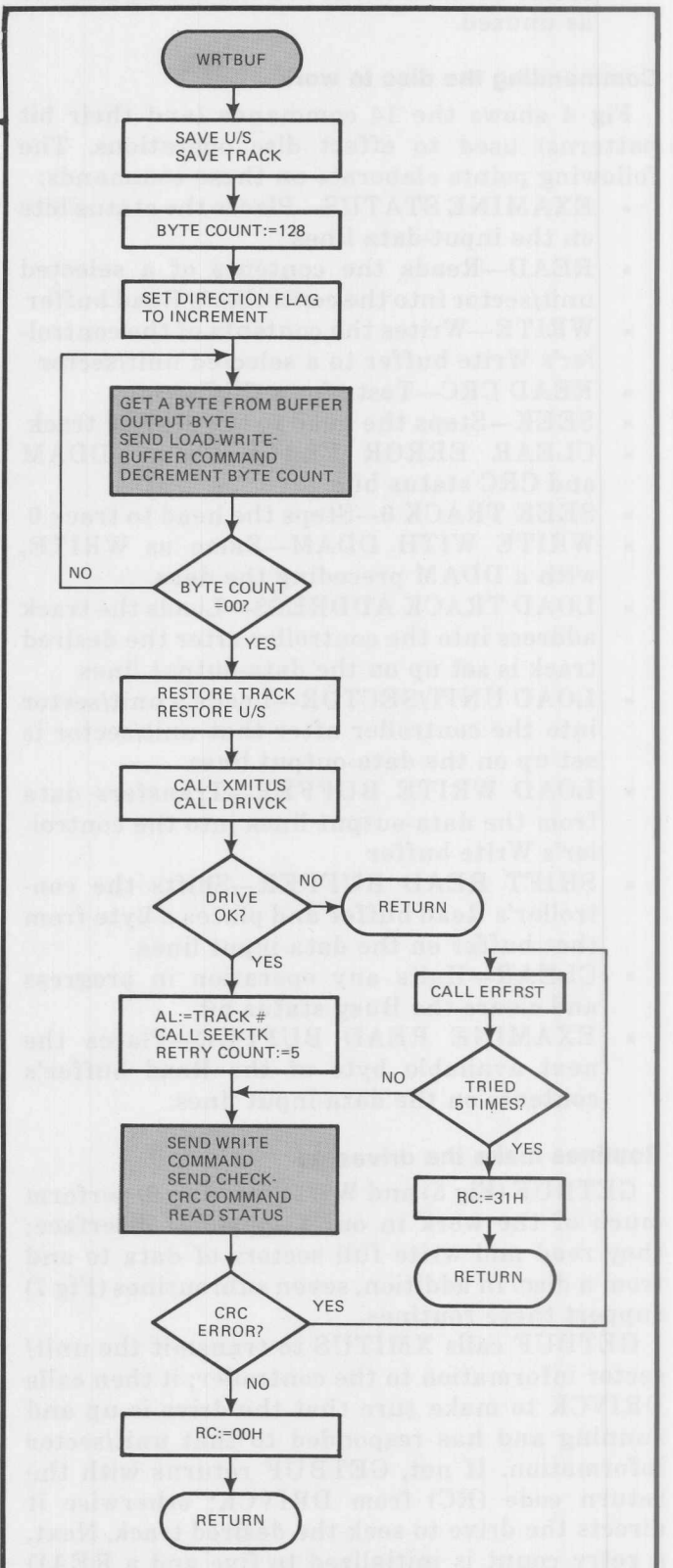


Fig 6—WRTBUF writes a full sector of data to a disc *from the Write buffer.*

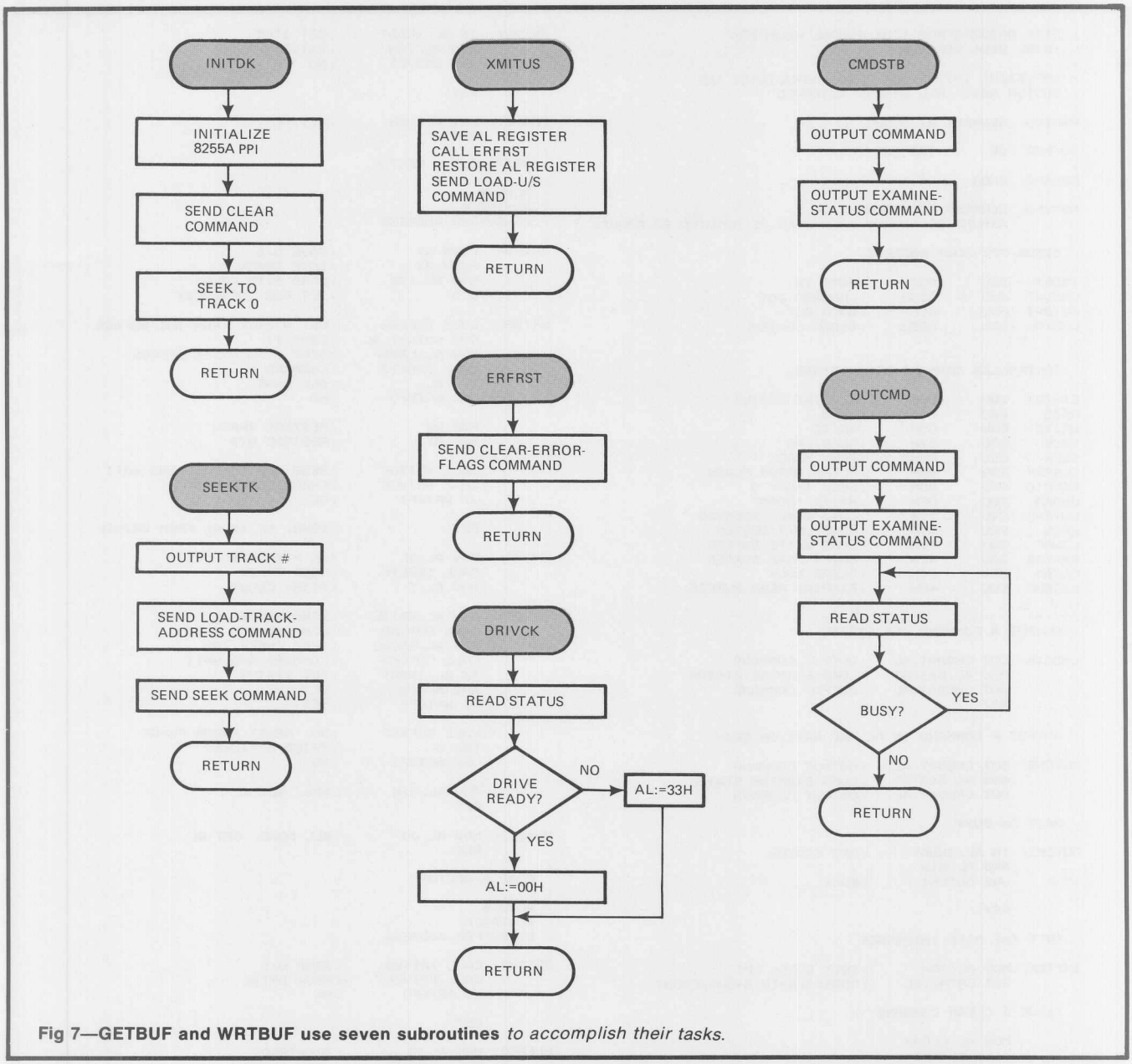


Fig 7—GETBUF and WRTBUF use seven subroutines *to accomplish their tasks.*

that the drive is up and running. If not, WRTBUF returns with the RC from DRIVCK; otherwise it issues a SEEK command for the desired track and sets the retry count to five. The routine issues a WRITE command to transfer the contents of the Write buffer to the disc and checks for a CRC error condition. If such an error occurs, the routine calls ERFRST to clear the flags in the controller and then retries the WRITE. If the WRITE succeeds within five tries, WRTBUF returns with an RC of 00; if it's unsuccessful, the RC is 31_H.

Executing from the monitor

To start up the system from the resident monitor in either the SDK-86 or iSBC 86/12, hit the Reset button to put the computer in a known state, and use the monitor GO command to jump to the initialization routine. Then load the unit/sector information in the AX register's AL section (as shown in **Fig 3**) and the track number in the BX register's BL section, point the SI register to the RAM location of the data to be written onto disc and point the DI register to the RAM's Read buffer. The next step is to jump (using the GO command) to the WRITE routine. When the monitor returns, check the SI register—it should increment by 80_H. Reset the AX and BX registers and jump to the READ routine. You can now check the Read-buffer location to observe the data that was on the disc.

EDN

```
; DISK DRIVERS FOR ICOM FRUGAL FLOPPIES
; INTEL 8086 VERSION
;
; COPYRIGHT 1979 BY HEMENWAY ASSOCIATES INC
; BOSTON MASS.  ALL RIGHTS RESERVED
;
RAMSEG  SEGMENT AT 300H
;
BUFFER  DB      128 DUP (0)
;
RAMSEG  ENDS
;
ROMSEG  SEGMENT AT 0200H
        ASSUME CS:ROMSEG,DS:RAMSEG,SS:NOTHING,ES:RAMSEG
;
; 8255A PPI PORT ADDRESS:
;
INDAT   EQU     0C8H    ;DATA IN
CMDDAT  EQU     0CAH    ;COMMAND OUT
OUTDAT  EQU     0CCH    ;DATA OUT
CONTRL  EQU     0CEH    ;8255A CONTROL
;
;
; CONTROLLER COMMAND DEFINITIONS:
;
EXSTAT  EQU     00H     ;EXAMINE STATUS
READ    EQU     03H     ;READ
WRITE   EQU     05H     ;WRITE
RDCRC   EQU     07H     ;READ CRC
SEEK    EQU     09H     ;SEEK TRACK
CLRERF  EQU     0BH     ;CLEAR ERROR FLAGS
SEEKTO  EQU     0DH     ;SEEK TRACK 0
WDDAM   EQU     0FH     ;WRITE "DDAM"
LDTRAD  EQU     11H     ;LOAD TRACK ADDRESS
LDUS    EQU     21H     ;LOAD UNIT/SECTOR
LDWBF   EQU     31H     ;LOAD WRITE BUFFER
SHFTRB  EQU     41H     ;SHIFT READ BUFFER
CLEAR   EQU     81H     ;CLEAR BUSY
EXRDBF  EQU     40H     ;EXAMINE READ BUFFER
;
;
; OUTPUT A COMMAND AND STROBE
;
CMDSTB: OUT CMDDAT,AL   ;OUTPUT COMMAND
        MOV AL,EXSTAT   ;LOAD EXAMINE STATUS
        OUT CMDDAT,AL   ;OUTPUT COMMAND
        RET
;
; OUTPUT A COMMAND IN AL AND WAIT ON BUSY
;
OUTCMD: OUT CMDDAT,AL   ;OUTPUT COMMAND
        MOV AL,EXSTAT   ;LOAD EXAMINE STATUS
        OUT CMDDAT,AL   ;OUTPUT COMMAND
;
; WAIT ON BUSY
;
OUTCM1: IN AL,INDAT     ;GET STATUS
        AND AL,01H
        JNZ OUTCM1      ;BUSY
;
        RET
;
; INIT THE DISK INTERFACE
;
INITDK: MOV AL,90H      ;INIT 8255A PPI
        OUT CNTRL,AL    ;MODE=0,A=IN,B=OUT,C=OUT
;
; ISSUE A CLEAR COMMAND
;
        MOV AL,CLEAR
        CALL OUTCMD
;
; SEEK TRACK 0
;
        MOV AL,SEEKTO
        CALL OUTCMD
        RET
;
; CLEAR ERROR FLAGS
;
ERFRST: MOV AL,CLRERF
        CALL CMDSTB
        RET
;
; SEEK TRACK IN AL
;
SEEKTK: OUT OUTDAT,AL   ;OUTPUT TRACK #
        MOV AL,LDTRAD   ;SEND LOAD TRACK
        CALL CMDSTB     ;COMMAND
        MOV AL,SEEK     ;SEND SEEK
        CALL OUTCMD     ;COMMAND AND WAIT
        RET
;
; TRANSMIT U/S IN AL
;
XMITUS: PUSH AX         ;SAVE U/S
        CALL ERFRST     ;CLEAR ERROR FLAGS
        POP AX          ;RESTORE U/S
        OUT OUTDAT,AL   ;SEND U/S
        MOV AL,LDUS     ;SEND LOAD U/S
        CALL CMDSTB     ;COMMAND
        RET

  CHECK DRIVE

DRIVCK: IN AL,INDAT     ;GET STAT
        AND AL,20H      ;DRIVE OK?
        JNZ DRIVC1      ;NO
;
        RET
;
DRIVC1: MOV AL,33H      ;SET RC
        RET
;
; WRITE OUT A SECTOR
;
; AL=U/S
; BL=TRACK
; SI=BUFFER ADDRESS
;
WRTBUF: PUSH AX         ;SAVE U/S
        PUSH BX         ;SAVE TRACK
        MOV BL,128      ;LOAD BYTE COUNT
        CLD             ;SET FOR INCREMENT
;
WRTBF0: LODS BUFFER     ;GET A BYTE FROM THE BUFFER
        OUT OUTDAT,AL   ;SEND IT
        MOV AL,LDWBF    ;SEND LOAD WRITE BUFFER
        CALL CMDSTB     ;COMMAND
        DEC BL          ;ALL DONE
        JNZ WRTBF0      ;NO
;
        POP BX          ;RESTORE TRACK
        POP AX          ;RESTORE U/S
;
        CALL XMITUS     ;SEND U/S COMMAND AND WAIT
        CALL DRIVCK     ;CHECK DRIVE
        JZ WRTBF1       ;OK
;
        RET             ;DOWN, RC IN AL FROM DRIVCK
;
WRTBF1: MOV AL,BL       ;AL:=TRACK #
        CALL SEEKTK     ;SEEK TRACK
        MOV BL,5        ;RETRY COUNT
;
WRTBF2: MOV AL,WRITE    ;SEND WRITE
        CALL OUTCMD     ;COMMAND AND WAIT
        MOV AL,RDCRC    ;SEND CHECK CRC
        CALL OUTCMD     ;COMMAND AND WAIT
        IN AL,INDAT     ;GET STATUS
        AND AL,08       ;OK?
        JZ WRTBF3       ;YES
;
        CALL ERFRST     ;NO, RESET ERROR FLAGS
        DEC BL          ;TRIED 5 TIMES?
        JNZ WRTBF2      ;NO
;
        MOV AL,31H      ;YES, SET RC
        RET
;
WRTBF3: MOV AL,00       ;ALL DONE, SET RC
        RET
;
; READ A SECTOR
;
; AL=U/S
; BL=TRACK
; DI=BUFFER ADDRESS
;
GETBUF: CALL XMITUS     ;SEND U/S
        CALL DRIVCK     ;CHECK DRIVE
        JZ GETBF0       ;OK
;
        RET
;
GETBF0: MOV AL,BL       ;AL:=TRACK
        CALL SEEKTK     ;SEEK TRACK
        MOV BL,5        ;RETRY COUNT
;
GETBF1: MOV AL,READ     ;SEND READ
        CALL OUTCMD     ;COMMAND AND WAIT
        IN AL,INDAT     ;GET STATUS
        AND AL,08       ;CRC ERROR?
        JZ GETBF2       ;NO
;
        CALL ERFRST     ;CLEAR ERROR FLAGS
        DEC BL          ;TRIED 5 TIMES?
        JNZ GETBF1      ;NO
;
        MOV AL,31H      ;YES, SET RC
        RET
;
GETBF2: MOV BL,128      ;INIT BYTE COUNT
        CLD             ;SET FOR INCREMENT
;
GETBF3: MOV AL,EXRDBF   ;SEND EXAMINE READ
        OUT CMDDAT,AL   ;BUFFER COMMAND
        IN AL,INDAT     ;GET A BYTE
        PUSH AX         ;SAVE IT
        MOV AL,SHFTRB   ;SEND STROBE READ
        CALL CMDSTB     ;BUFFER COMMAND
        POP AX          ;RESTORE BYTE
        STOS BUFFER     ;STORE BYTE IN BUFFER
        DEC BL          ;ALL DONE?
        JNZ GETBF3      ;NO
;
        MOV AL,00       ;YES, SET RC
        RET
;
ROMSEG  ENDS
        END
```

Increase 8086 throughput by using interrupts

The proper use of a single-board computer's hardware and software can provide up to 64 levels of prioritized interrupts. Here's how such interrupts work and how to deal with them.

Jack Hemenway and **Edward Teja,**
Associate Editors

The process of making a 16-bit μC like the iSBC 86/12 talk to the outside world involves providing it with the means of servicing more than one I/O device. For although our prior discussion of a floppy-disc interface for the 8086 (EDN, March 20, pg 118) assumed that the processor was waiting around for the floppy disc to complete its operation, the processor could in fact have other work to do. There's no valid reason why the processor can't do this work while the disc completes its job—provided the processor has a means of knowing when the disc (or other I/O device) has completed the task.

Pause for an interruption

The most common servicing method involves polling every I/O device periodically: The processor tests each device, in a predetermined sequence, to see if that device needs servicing. But although polling serves well in certain applications (such as dealing with multiple terminals in a time-sharing network), it proves inefficient for use with μCs; the polling cycle reduces the

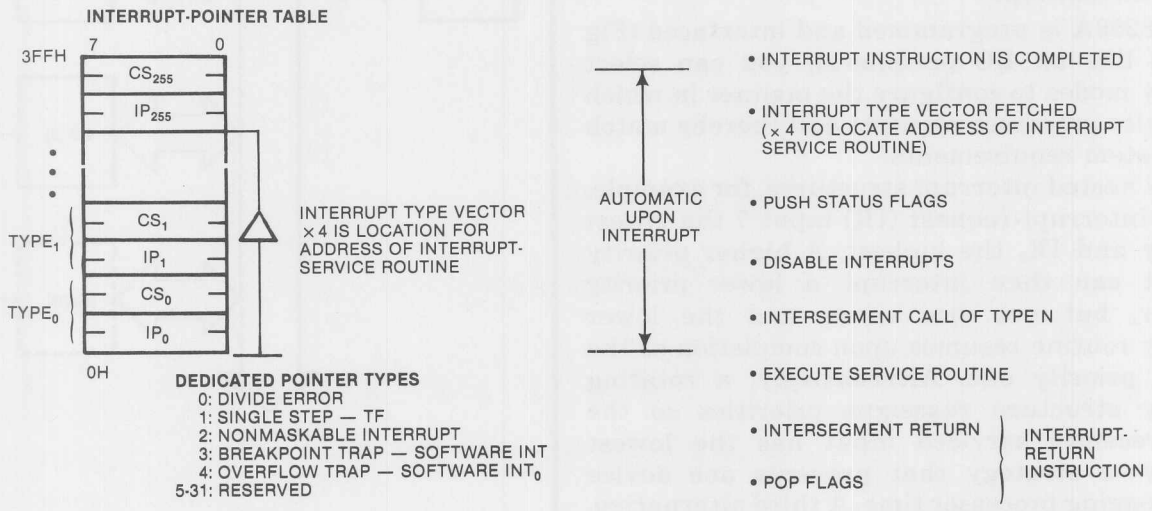


Fig 1—An interrupt sequence *permits program execution to resume exactly where it was interrupted, after completion of a service routine.*

Vectors point the processor to the interrupt handler

number of tasks that the computer can assume.

An optimum multi-I/O system thus allows the processor to execute its main program without pause, stopping *only* when a device requests its attention. This request takes the form of an asynchronous input termed an interrupt, which causes the processor to complete its current instruction, service the interrupt, then resume exactly where it left off (**Fig 1**). A specialized set of instructions, called the interrupt handler, assumes the task of actually servicing the interrupt. These instructions constitute a routine; an example is one of the disc-I/O routines.

The 8086 provides up to 64 interrupt levels; each group of eight levels comes from one 8259A programmable interrupt controller (PIC). Functioning as a manager in an interrupt-driven environment, this device provides the hardware support necessary to accept interrupt requests from peripheral equipment, determine the requests' priority, ascertain whether a higher priority interrupt is currently being serviced, and issue the interrupt to the μP.

How does the processor find out which interrupt handler to use to satisfy a particular request? One simple method, termed vectored interrupt, points the μP's program counter (PC) to the correct address by having the PIC consult a table of such addresses. This address, also termed a vector or vectoring data, then goes to the PC at the PIC's direction. In the case of the 8086, each table entry comprises a value for the CS and IP registers.

PICing an interrupt

The 8259A is programmed and interfaced (**Fig 2**) just like an I/O peripheral; you can select priority modes to configure the manner in which the device processes requests and thereby match it to system requirements.

Fully nested interrupt structures, for example, assign interrupt-request (IR) input 7 the lowest priority and IR_0 the highest. A higher priority request can then interrupt a lower priority handler, but not vice versa, and the lower priority routine resumes upon completion of the higher priority one. Alternatively, a rotating priority structure reassigns priorities so the most recently serviced input has the lowest priority—a strategy that prevents one device from hogging processor time. A third alternative, termed specific priority, provides for altering priorities to suit the needs of an executed program.

In practice, you can combine these operation modes, dynamically selecting the appropriate one under software control. A further modification permits the masking of individual interrupts—an approach that prevents a particular device from causing an interrupt regardless of its priority.

Initializing the system

To further understand the PIC's role in interrupt handling, you must understand the part that the 8086's monitor plays in preparing the system for interrupts.

The iSBC 86/12 monitor program resides in EPROM; **Fig 3** shows a memory map of the μC with this monitor program. Note that the top 8k bytes contain the monitor itself, while the lowest 384 bytes contain the monitor and user stacks, monitor data and interrupt vectors. This arrangement illustrates the PIC's relationship to the vector-address table.

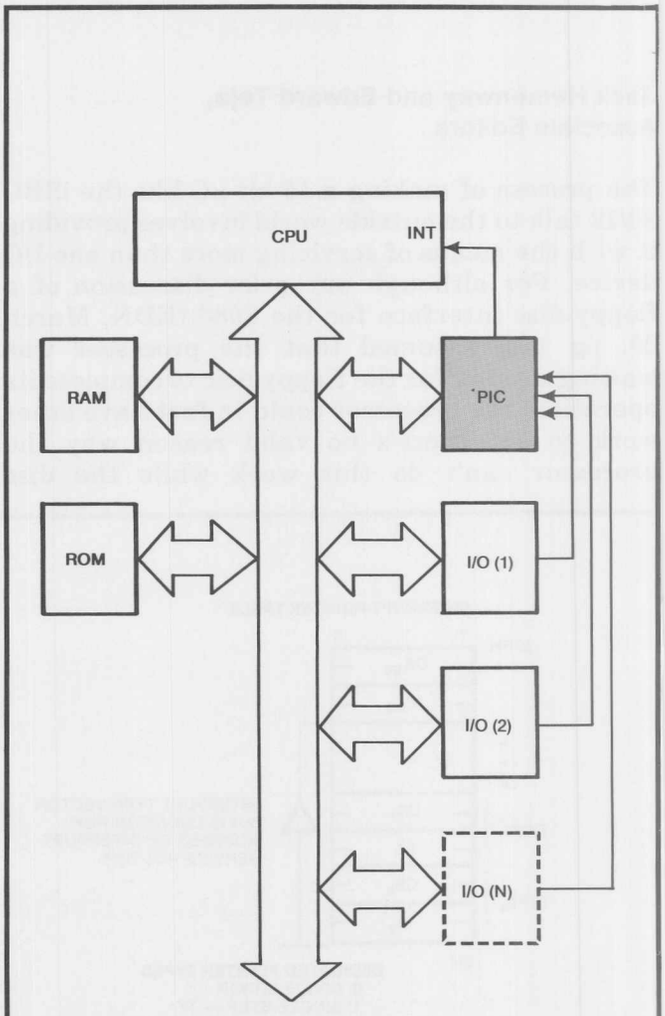


Fig 2—A programmable interrupt controller (PIC) *handles up to eight interrupts; you can cascade such devices to provide up to 64 vectored priority interrupts with no additional circuitry.*

Warning for the hasty

When finalizing a design based on the 8086, beware of a few problem areas not readily noticeable. The μP's documentation is currently a bit weak; generating this article required synthesizing information from almost every manual Intel can provide in support of the system, then spending a fair amount of time in phone calls verifying that we understood what those manuals almost said, or didn't say.

For example, the section on the functioning of the 8259A in the "iSBC 86/12 Single-Board-Computer Hardware Reference Manual" is misleading; go straight to the "MCS-86 User's Manual."

One interrupt-oriented difficulty concerns the PL/M-86 compiler. The iSBC 86/12 is shipped with the counter 0 connected to IR_2; this procedure isn't necessarily bad, but the 8253 counter isn't initialized by the monitor. Power-up can thus produce a continuous square wave, generating unwanted interrupts.

Also, note that when emerging from a critical section (a portion of a program that's protected from interrupts) the compiler enables the interrupts regardless of their status when it entered that critical section. (Some applicable rules for treating critical sections appear in the Software Note in EDN, May 5, pg 88.)

Resetting the monitor sets the μP's segment registers, IP and flags to zero; it sets the SP to $01C0_H$, providing 64 bytes for the user stack. More important to this discussion, though, the monitor sets the single-step, 1-byte instruction trap and nonmaskable interrupt vectors to monitor-entry points and assigns the eight 8259A PIC vectors, starting at 80_H, to point to various monitor locations. The 8259A is programmed to the fully nested mode with level 0 as the highest one; all

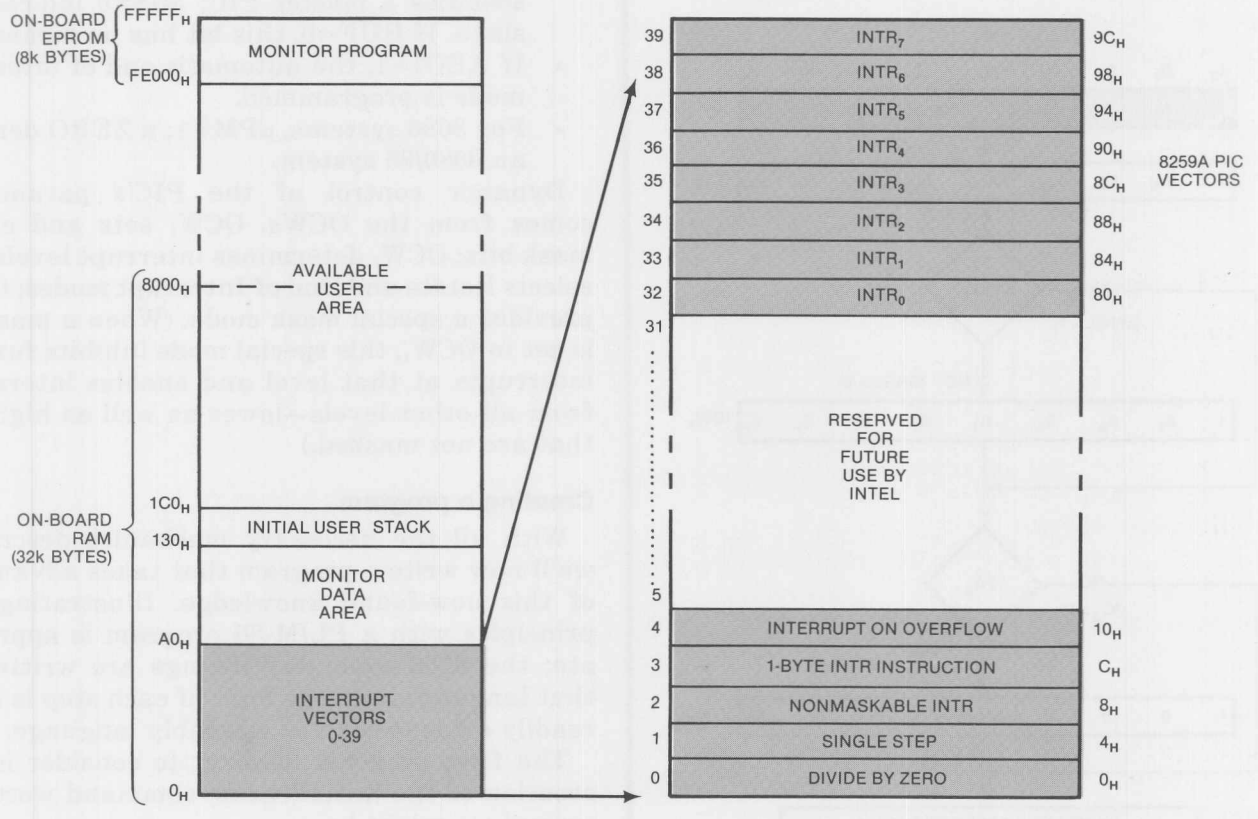


Fig 3—The memory map *for an iSBC 86/12 μC board shows the location of the interrupt vectors.*

Control interrupt parameters with operation-control words

interrupts are enabled.

The initialization sequence conditions the system so that when an interrupt occurs, control passes to the monitor from whatever program is executing. The sequence acknowledges the interrupt, displaying the interrupt level, CS and IP registers and next-instruction byte on the system console.

Because the PIC is programmable, you needn't leave it in the mode that the monitor places it in. Your programs can control the device's operation via its initialization-command words (ICWs) and operation-command words (OCWs). Each 8259A in the system (remember that there can be as many as eight) requires ICWs to tell it

- If there are any other PICs in the system, and how they are connected
- The starting addresses of the service routines
- Whether the service routines are four or eight bytes apart.

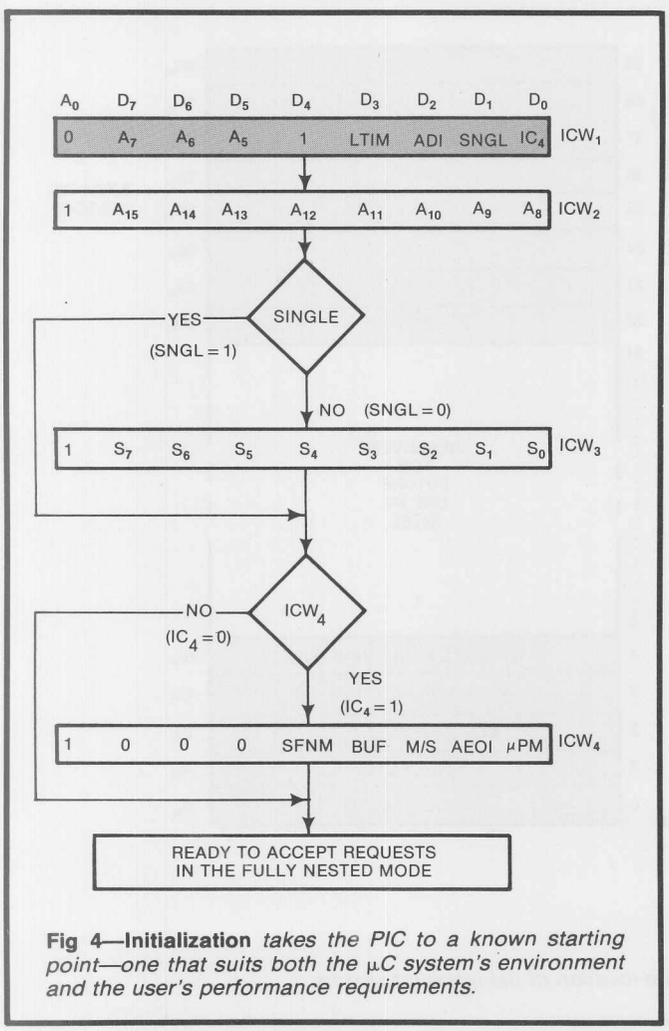


Fig 4—Initialization *takes the PIC to a known starting point—one that suits both the μC system's environment and the user's performance requirements.*

Setting A_0=0 and D_4=1 identifies a command as ICW_1 and begins the initialization sequence which must be completed before interrupts can be processed. **Fig 4** illustrates the initialization process' flow and the command-word formats.

Each IR input is associated with an address in memory. You insert address bits A_{15} through A_{11} in the five most significant bits of the vectoring byte (ICW_2); the PIC then sets the three least significant bits according to the interrupt level. Address bits A_{10} through A_5, and the address interval (ADI), serve 8080 systems and are thus ignored in 8086 configurations. Additionally,

- If LTIM (D_3 of ICW_1)=1, the PIC operates in the level-interrupt mode; edge-detect logic is disabled.
- If SNGL=1, the PIC is the only one in the system, and no ICW_3 is issued.
- If IC_4 is set, ICW_4 must be read.

If SNGL≠1, you must inform each PIC, via ICW_3, whether it is the master or a slave. ICW_3 is reserved for specifying that linkage.

ICW_4 specifies the following information:

- If SFNM=1, the fully nested mode is programmed.
- If BUF=1, the buffered mode is programmed.
- If the buffered mode is selected, M/S=1 specifies a master PIC; M/S=0 indicates a slave. If BUF=0, this bit has no meaning.
- If AEOI=1, the automatic end of interrupt mode is programmed.
- For 8086 systems, μPM=1; a ZERO denotes an 8080/85 system.

Dynamic control of the PIC's parameters comes from the OCWs. OCW_1 sets and clears mask bits; OCW_2 determines interrupt levels and selects Rotate and End of Interrupt modes; OCW_3 provides a special mask mode. (When a mask bit is set in OCW_1, this special mode inhibits further interrupts at that level and enables interrupts from all other levels—lower as well as higher—that are not masked.)

Creating a program

With all the necessary mechanics described we'll now write a program that takes advantage of this new-found knowledge. Illustrating the principles with a PL/M-86 program is appropriate; the 8086's monitor listings are written in that language, and the logic of each step is more readily evident than in assembly language.

The first program element to consider is the creation of the initialization-command words. A typical set might be

```
DECLARE /*8259A INTERRUPT CONTROLLER*/
IC$PORTA LITERALLY 'OCOH',/* PORT A */
IC$PORTB LITERALLY 'OC2H', /* PORT B */
IC$ICW1   LITERALLY '17H',   /* INIT ICW1 */
IC$ICW2   LITERALLY '20H',   /* INIT ICW2 */
```

```
IC$ICW4   LITERALLY '1DH',   /* INIT ICW4 */
IC$MASK   LITERALLY 'OOH',   /* INTERRUPT
                                MASK */
```

With command words defined, we next provide for the PIC's initialization:

```
OUTPUT(IC$PORTA)=IC$ICW1;
OUTPUT(IC$PORTB)=IC$ICW2;
OUTPUT(IC$PORTB)=IC$ICW4;
OUTPUT(IC$PORTB)=IC$MASK;
```

A set of individual procedures then provides the interrupt handling for specific devices; each procedure describes a unique set of activities necessary to properly respond to a particular interrupt. An interrupt could, for example, indicate that the input buffer from a keyboard is full. The exact code required to react to this situation depends on the actual design and the hardware's requirements. The procedure, then, requires selecting some interrupt, such as interrupt-request number 32—$INTR_0$ interrupt input on the PIC. Similarly, number 33 is $INTR_1$ (see **Fig 3**), which represents the address that would point to the proper code. A user would then write that code, compiling it as a procedure:

```
IHANDLER$32:    PROCEDURE INTERRUPT 32;
    ...
    /* code to process this interrupt */
    ...
    ...
    RETURN;
END IHANDLER$32;
```

Each handler would have its own procedure, although it might not be necessary to use all eight interrupt requests. (In such cases, the monitor's initialization takes care of those extra requests.) Theoretically, no interrupts other than those designed for occur, but when the inevitable no-reason-for-it interrupt does happen, the monitor fields it and returns a statement to that effect.

Loading the table

Upon completion of the monitor initialization, the vector table always points to the monitor. How does the system know how to find the necessary procedure? As part of its normal procedure, the loader program sets a vector into the table that points to the handler routine corresponding to the interrupt number (32 in our example).

The net result of this effort is a smooth-running system that appears capable of performing several tasks simultaneously. It isn't, of course. Not with a single processor. You'll have to consider multiprocessing to obtain performance like that. **EDN**

U.S. AND CANADIAN SALES OFFICES

3065 Bowers Avenue
Santa Clara, California 95051
Tel: (408) 987-8080
TWX: 910-338-0026
TELEX: 34-6372

ALABAMA
Intel Corp.
3322 S. Parkway, Ste. 71
Holiday Office Center
Huntsville 35802
Tel: (205) 883-2430

†Pen-Tech Associates, Inc.
Holiday Office Center
3322 Memorial Pkwy., S.W.
Huntsville 35801
Tel: (205) 881-9298

ARIZONA
Intel Corp.
8650 N. 35th Avenue, Suite 101
Phoenix 85021
Tel: (602) 242-7205

†BFA
4426 North Saddle Bag Trail
Scottsdale 85251
Tel: (602) 994-5400

CALIFORNIA
Intel Corp.
7670 Opportunity Rd.
Suite 135
San Diego 92111
Tel: (714) 268-3563

Intel Corp.*
1651 East 4th Street
Suite 105
Santa Ana 92701
Tel: (714) 835-9642
TWX: 910-595-1114

Intel Corp.*
15335 Morrison
Suite 345
Sherman Oaks 91403
(213) 986-9510
TWX: 910-495-2045

Intel Corp.*
3375 Scott Blvd.
Santa Clara 95051
Tel: (408) 987-8086
TWX: 910-339-9279
TWX: 910-338-0255

Earle Associates, Inc.
4617 Ruffner Street
Suite 202
San Diego 92111
Tel: (714) 278-5441

Mac-I
2576 Shattuck Ave.
Suite 4B
Berkeley 94704
Tel: (415) 843-7625

Mac-I
P.O. Box 1420
Cupertino 95014
Tel: (408) 257-9880

Mac-I
P.O. Box 8763
Fountain Valley 92708
Tel: (714) 839-3341

Mac-I
110 Sutter Street
Suite 715
San Francisco 94104
Tel: (415) 982-3673

Mac-I
20121 Ventura Blvd., Suite 240E
Woodland Hills 91364
Tel: (213) 347-5900

COLORADO
Intel Corp.*
6000 East Evans Ave.
Bldg. 1, Suite 260
Denver 80222
Tel: (303) 758-8086
TWX: 910-931-2289

†Westek Data Products, Inc.
25921 Fern Gulch
P.O. Box 1355
Evergreen 80439
Tel: (303) 674-5255

Westek Data Products, Inc.
1322 Arapahoe
Boulder 80302
Tel: (303) 449-2620

CONNECTICUT
Intel Corp.
Peacock Alley
1 Padanaram Road, Suite 146
Danbury 06810
Tel: (203) 792-8366
TWX: 710-456-1199

FLORIDA
Intel Corp.
1001 N.W. 62nd Street, Suite 406
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407

FLORIDA (cont.)
Intel Corp.
5151 Adanson Street, Suite 203
Orlando 32804
Tel: (305) 628-2393
TWX: 810-853-9219

†Pen-Tech Associates, Inc.
201 S.E. 15th Terrace, Suite K
Deerfield Beach 33441
Tel: (305) 421-4989

†Pen-Tech Associates, Inc.
111 So. Maitland Ave., Suite 202
P.O. Box 1475
Maitland 32751
Tel: (305) 645-3444

GEORGIA
Pen Tech Associates, Inc.
Cherokee Center, Suite 21
627 Cherokee Street
Marietta 30060
Tel: (404) 424-1931

ILLINOIS
Intel Corp.*
900 Jorie Boulevard
Suite 220
Oakbrook 60521
Tel: (312) 325-9510
TWX: 910-651-5881

First Rep Company
9400-9420 W. Foster Avenue
Chicago 60656
Tel: (312) 992-0830
TWX: 910-227-4927

Technical Representatives
1502 North Lee Street
Bloomington 61701
Tel: (309) 829-8080

IOWA
Technical Representatives, Inc.
St. Andrews Building
1930 St. Andrews Drive N.E.
Cedar Rapids 52405
Tel: (319) 393-5510

KANSAS
Intel Corp.
9393 W. 110th St., Ste. 265
Overland Park 66210
Tel: (913) 642-8080

Technical Representatives, Inc.
8245 Nieman Road, Suite #100
Lenexa 66214
Tel: (913) 888-0212, 3, & 4
TWX: 910-749-6412

KENTUCKY
Lowry & Associates Inc.
P.O. Box 5127
Lexington 40555
Tel: (606) 273-3771

MARYLAND
Intel Corp.*
7257 Parkway Drive
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

Glen White Associates
57 W. Timonium Road, Suite 307
Timonium 21093
Tel: (301) 252-6360

†Mesa Inc.
11900 Parklawn Drive
Rockville 20852
Tel: Wash. (301) 881-8430
Balto. (301) 792-0021

MASSACHUSETTS
Intel Corp.*
27 Industrial Ave.
Chelmsford 01824
Tel: (617) 667-8126
TWX: 710-343-6333

EMC Corp.
381 Elliot Street
Newton 02164
Tel: (617) 244-4740

MICHIGAN
Intel Corp.*
26500 Northwestern Hwy.
Suite 401
Southfield 48075
Tel: (313) 353-0920
TWX: 910-420-1212
TELEX: 2 31143

†Lowry & Associates, Inc.
135 W. North Street
Suite 4
Brighton 48116
Tel: (313) 227-7067

MINNESOTA
Intel Corp.
7401 Metro Blvd.
Suite 355
Edina 55435
Tel: (612) 835-6722
TWX: 910-576-2867

†Dytek North
1821 University Ave.
Room 163N
St. Paul 55104
Tel: (612) 645-5816

MISSOURI
Technical Representatives, Inc.
320 Brookes Drive, Suite 104
Hazelwood 63042
Tel: (314) 731-5200
TWX: 910-762-0618

NEW JERSEY
Intel Corp.*
1 Metroplaza Office Bldg.
505 Thornall St.
Edison 08817
Tel: (201) 494-5040
TWX: 710-480-6238

NEW MEXICO
BFA Corporation
P.O. Box 1237
Las Cruces 88001
Tel: (505) 523-0601
TWX: 910-983-0543

BFA Corporation
3705 Westerfield, N.E.
Albuquerque 87111
Tel: (505) 292-1212
TWX: 910-989-1157

NEW YORK
Intel Corp.*
350 Vanderbilt Motor Pkwy.
Suite 402
Hauppauge 11787
Tel: (516) 231-3300
TWX: 510-227-6236

Intel Corp.
80 Washington St.
Poughkeepsie 12601
Tel: (914) 473-2303
TWX: 510-248-0060

Intel Corp.
2255 Lyell Avenue
Lower Floor East Suite
Rochester 14606
Tel: (716) 328-7340
TWX: 510-253-3841

†Measurement Technology, Inc.
159 Northern Boulevard
Great Neck 11021
Tel: (516) 482-3500

T-Squared
4054 Newcourt Avenue
Syracuse 13206
Tel: (315) 463-8592
TWX: 710-541-0554

T-Squared
2 E. Main
Victor 14564
Tel: (716) 924-9101
TWX: 510-254-8542

NORTH CAROLINA
†Pen-Tech Associates, Inc.
1202 Eastchester Dr.
Highpoint 27260
Tel: (919) 883-9125

Glen White Associates
4009 Barrett Dr.
Raleigh 27609
Tel: (919) 787-7016

OHIO
Intel Corp.*
8312 North Main Street
Dayton 45415
Tel: (513) 890-5350
TWX: 810-450-2528

Intel Corp.*
Chagrin-Brainard Bldg. #210
28001 Chagrin Blvd.
Cleveland 44122
Tel: (216) 464-2736

Lowry & Associates Inc.
1440 Snow Road
Suite 216
Cleveland 44134
Tel: (216) 398-0506

†Lowry & Associates, Inc.
1524 Marsetta Drive
Dayton 45432
Tel: (513) 429-9040

†Lowry & Associates, Inc.
1050 Freeway Dr., N.
Suite 209
Columbus 43229
Tel: (614) 436-2051

OREGON
Intel Corp.
10700 S.W. Beaverton
Hillsdale Highway
Suite 324
Beaverton 97005
Tel: (503) 641-8086

PENNSYLVANIA
Intel Corp.*
275 Commerce Dr.
200 Office Center
Suite 300
Fort Washington 19034
Tel: (215) 542-9444
TWX: 510-661-2077

†Lowry & Associates, Inc.
Seven Parkway Center
Suite 455
Pittsburgh 15520
Tel: (412) 922-5110

†Q.E.D. Electronics
300 N. York Road
Hatboro 19040
Tel: (215) 674-9600

TEXAS
Intel Corp.*
2925 L.B.J. Freeway
Suite 175
Dallas 75234
Tel: (214) 241-9521
TWX: 910-860-5487

Intel Corp.*
6420 Richmond Ave.
Houston 77057
Tel: (713) 784-3400

Industrial Digital Systems Corp.
5925 Sovereign
Suite 101
Houston 77036
Tel: (713) 988-9421

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

VIRGINIA
Glen White Associates
Route 2, Box 193
Charlottesville 22901
Tel: (804) 295-7686

Glen White Associates
P.O. Box 10186
Lunchburg 24506
Tel: (804) 384-6920

Glen White Associates
Rt. #1, Box 322
Colonial Beach 22442
Tel: (804) 224-7764

WASHINGTON
Intel Corp.
Suite 114 Bldg. 3
1603 116th Ave. N.E.
Bellevue 98005
Tel: (206) 453-8086

WISCONSIN
Intel Corp.
4369 S. Howell Ave.
Milwaukee 53207
Tel: (414) 747-0789

CANADA
Intel Semiconductor Corp.*
Suite 233, Bell Mews
39 Highway 7, Bells Corners
Ottawa, Ontario K2H 8R2
Tel: (613) 829-9714
TELEX: 053-4115

Intel Semiconductor Corp.
6205 Airport Rd.
Bldg. B, Suite 205
Mississauga, Ontario
L4V 1E3
Tel: (416) 671-0611
TELEX: 06983574

Multilek, Inc.*
15 Grenfell Crescent
Ottawa, Ontario K2G 0G3
Tel: (613) 226-2365
TELEX: 053-4585

Multilek, Inc.
Toronto
Tel: (416) 245-6422

Multilek, Inc.
Montreal
Tel: (514) 481-1350

*Field application location
†These representatives do not offer Intel Components, only boards and systems.

98-01014-01